Hood: A User-level Thread Library for Multiprogramming Multiprocessors

by

Dionysios P. Papadopoulos, B.S.

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Arts

The University of Texas at Austin

August 1998

Hood: A User-level Thread Library for Multiprogramming Multiprocessors

Approved by
Supervising Committee:

Supervisor _____

Robert D. Blumofe

_____

Calvin Lin

In memory of
my grandmother Kalliopi Anastasopoulou
and my uncle Yannis Skliris.

Acknowledgments

I would like to thank my advisor, Professor Robert Blumofe, who gave me the opportunity to work in such an interesting area. I faced a plethora of challenges and he was always there to help me, work with me and give me advice with a lot of patience. Through my collaboration with Bobbby, I learned a lot about conducting research and how to improve myself, by being more methodical, productive, and organized. Above all, I really had a great time.

I would also like to thank Professor Calvin Lin for his comments and suggestions as a reader of this thesis.

I am grateful to Umut Acar and Emery Berger who were always willing to listen to any problems and provide a lot of helpful ideas, comments and suggestions. I also want to express my appreciation to Boyd Merworth for all of his technical assistance with the running of the experiments.

Several people outside UT have also contributed to this work. Keith Randall of MIT found a bug in an early version of the the non-blocking implementation of the work-stealing algorithm and Mark Moir of The University of Pittsburgh, suggested ideas that lead us to a correct algorithm.

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under Grant F30602-97-1-0150 from the U.S. Air Force Research Laboratory. Multiprocessor computing facilities were provided through a generous donation by Sun Microsystems, Inc.

I would especially like to express my gratitude to my cousin Alex for his help and support during the two years that we were together in Austin. Finally, I would like to thank my family, my grandmother Sophia, my brothers Anastasis and Alexandros, and especially my parents Periklis and Flora. I would not have the opportunity to be here and finish this thesis without their love and support.

DIONYSIOS P. PAPADOPOULOS

The University of Texas at Austin
August 1998

Hood: A User-level Thread Library for Multiprogramming Multiprocessors

Dionysios P. Papadopoulos, M.A.
The University of Texas at Austin, 1998

Supervisor: Robert D. Blumofe

This thesis presents the design and implementation of Hood, a C++ user-level threads library for parallel programming targeted for shared-memory multiprocessors. Hood supports the abstraction of user-level threads, and it schedules those threads onto processes using a non-blocking implementation of the work-stealing algorithm. We show that our non-blocking implementation of the work-stealing algorithm delivers efficient performance under multiprogramming without any need for kernel-level resource management, such as coscheduling. With this implementation, the execution time of a computation running with arbitrarily many processes on arbitrarily many processors can be modeled as a simple function of work and critical-path length. This model holds even when the processes run on a set of processors that arbitrarily grows and shrinks over time.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As small-scale multiprocessors make their way onto desktops, the high-performance parallel applications that run on these machines will have to live alongside other applications, such as editors and web browsers. Similarly, users expect multiprocessor compute servers to support multiprogrammed work loads that include parallel applications. Unfortunately, unless parallel applications are coscheduled [42] or subject to process control [48], they display poor performance in such multiprogrammed environments [10, 17, 19, 20, 27]. As an alternative to coscheduling or process control, Hood employs dynamic, user-level, thread scheduling and achieves efficient performance under multiprogramming. Hood is a C++ user-level threads library for parallel programming targeted for shared-memory multiprocessors. It supports the abstraction of user-level threads, and it schedules those threads onto processes using a non-blocking implementation of the work-stealing algorithm.

Throughout this thesis we shall use the word "process" to denote a kernel-scheduled entity, and we shall assume that all processes belonging to the same executing program can share memory and synchronize through the use of synchronization variables. Such processes are often referred to as "light-weight processes" or "kernel threads." We shall reserve the word "thread" to denote a user-level task that

is scheduled by a user-level library. Our implementation has two scheduling levels: The Hood user-level library schedules threads onto a fixed collection of processes, while below the operating-system kernel schedules processes onto a fixed collection of processors. Our scheduler utilizes efficiently whatever set of processors the kernel scheduler happens to give it, even if the kernel scheduler gives it fewer processors than it has processes and even if that set of processors grows and shrinks over time.

We have developed and evaluated a simple performance model based on "work" and "critical-path length" that characterizes accurately the performance of parallel applications that use the Hood implementation of the non-blocking work stealer. In fact, this performance model is based on an analytical bound that has been proven to hold in a model where the kernel-level scheduling is actually performed by an adversary [9]. Thus, our model is extraordinarily robust. Moreover, we have developed a collection of prototype applications to prove the efficiency of our library. All of our applications have been written in C++ on top of Hood and vary from matrix computations to n-body simulations and ray tracing. We show that Hood delivers efficient performance under multiprogramming. For all the applications, we observe linear speedup whenever the number of processes is small relative to the average parallelism.

Our implementation of Hood is built on top of the Solaris thread library [47], and it implements each process as a Solaris Light-Weight Process (LWP). It also uses the atomic casxa [46] (64-bit compare and swap instruction of the SPARC v9 architecture) to implement a non-blocking version of the work-stealing algorithm. All experiments were performed on a Sun Ultra Enterprise 5000 with 8 167-Mhz UltraSPARC processors running Solaris 2.5.1.

1.1 The problem with static partitioning

Before considering Hood's dynamic thread scheduling, we first review a well-known performance anomaly that occurs when parallel programs use a static partitioning of the work [33, pages 284–285]. In the simplest case when such a program executes, it creates some number P of processes, where typically P is selected by a command-line argument, and each process performs a $1/P$ fraction of the total work. Let T_1 denote the ***work*** of the computation, which we define as the execution time with $P = 1$ process. Using $P \geq 1$ processes, each process performs T_1/P work, and if the overhead of creating and synchronizing these processes is small compared to the T_1/P work per process, then we can hope that the execution time T_P will be given by $T_P = T_1/P$, thereby giving a ***speedup*** of $T_1/T_P = P$. Of course, this aspiration assumes that we have at least P processors on which to execute the program.

In a multiprogrammed environment, we might find that the actual number P_A of processors on which our program runs is smaller than the number P of processes, and in this case we cannot hope to achieve a speedup of P. Note that we always have $P_A \leq P$, because a program cannot run on more processors than it has processes. Thus, in a multiprogrammed environment, we can reasonably aspire to achieve an execution time of $T_P = T_1/P_A$, thereby giving a speedup of $T_1/T_P = P_A$ — that is, ***linear speedup*** — and a (processor) ***utilization*** of $T_1/(P_A T_P) = 1.0$. Unfortunately, for some problem inputs, our statically partitioned applications do not come close to fulfilling this aspiration unless we have $P_A = P$, effectively a single-user, dedicated machine.

Figure 1.1 shows the measured speedup of several statically partitioned applications for different numbers P of processes. More information about these applications is given in Table 2.1, and various characteristics for each of these applications, including the value of T_1, are given in Table 2.2. The applications are run on a dedicated machine with $P_M = 8$ processors, so the actual number P_A of processors

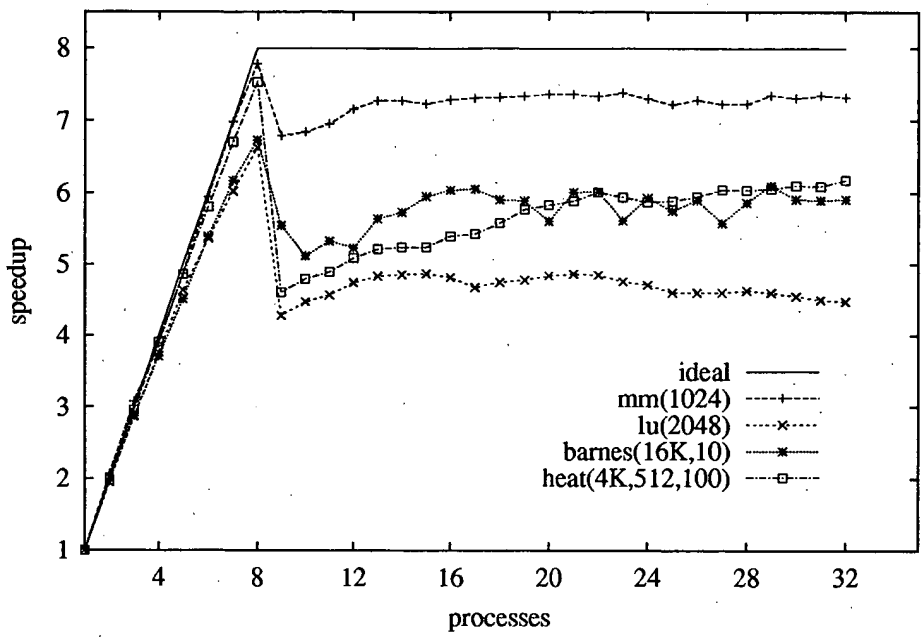

Figure 1.1: Measured speedup for statically partitioned applications. Speedup is plotted as a function of the number P of processes used when run on a dedicated 8-processor machine.

used is given by $P_A = \min\{P_M, P\} = \min\{8, P\}$. Observe that when we have $P \leq 8$, we have $P_A = P$, and all four applications come reasonably close to the ideal linear speedup. On the other hand, when we have $P > 8$, we have $P_A < P$, and performance drops off dramatically. In fact, the worst case is when we are off by only 1 — that is, when $P = 9 = P_A + 1$. In this case, the P_A processors begin by executing $P - 1$ of the processes, all of which complete in time T_1/P. Then, one of the processors executes the one remaining process, which also completes in time T_1/P. Thus, we have an execution time of $T_P = 2(T_1/P) = 2T_1/(P_A + 1)$, thereby giving a speedup of $T_1/T_P = (P_A + 1)/2 \approx P_A/2$ and an utilization of $T_1/(P_A T_P) = (P_A + 1)/(2P_A) \approx 0.5$ — roughly half the desired speedup and uti-

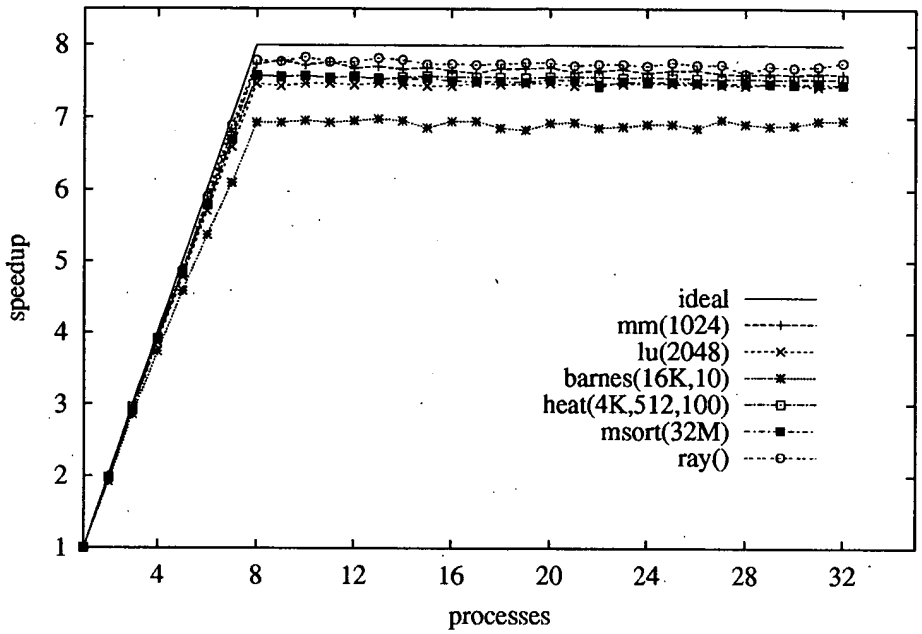

Figure 1.2: Measured speedup for Hood applications. Speedup is plotted as a function of the number P of processes used when run on a dedicated 8-processor machine.

lization.

The traditionally proposed solution to this problem is to use a number P of processes that is significantly greater than the number P_M of machine processors, so that we are guaranteed to have $P \gg P_A$ [33, page 285]. Indeed, using extra processes can improve the load imbalance, but as we see in Figure 1.1, it does not solve the problem. As P grows, the overhead of creating and synchronizing the processes grows and the work per process T_1/P shrinks. For sufficiently large values of T_1, this problem will not occur, because the time slicing divides each process into smaller pieces and fixes the load imbalance. Ultimately, however, this observation cannot console us. We want our applications to perform well for all input problems.

As an alternative to static partitioning, Hood employs a user-level thread

scheduler that dynamically assigns the application's work to its processes. In employing Hood, an application partitions its work into threads, where the amount of work in each thread and the number of threads is completely independent of the number of processors or processes. The partitioning into threads is determined by the amount of parallelism in the parallel algorithm being used. For example, in a divide-and-conquer algorithm in which the recursive subproblems can be solved in parallel, a separate thread is created for each recursive call. Thus, Hood applications may create millions of threads. Because the threads are created and synchronized at user-level, only a small amount of work per thread is need to amortize the cost of creating and synchronizing the myriad threads. Moreover, because Hood assigns threads to processes dynamically, applications deliver linear speedup even under multiprogramming. If we have $P_A < P$, then some processes will get less processor time than others (or maybe no processor time at all), but such processes will simply be assigned fewer (or no) threads to execute.

The result of Hood is performance as shown in Figure 1.2. Here we have performed the same experiment as in Figure 1.1 with the same applications and a couple more, but now the applications are recoded to use Hood. We observe that our applications now come very close to linear speedup across a very wide range of numbers P of processes, including the cases when we have $P_A < P$. Moreover, as we document in Chapter 2, we have not sacrificed any performance in the cases when we have $P_A = P$.

1.2 Contributions of this thesis

This thesis presents the design and implementation of Hood, a C++ user-level threads library for parallel programming targeted for shared-memory multiprocessors that delivers efficient performance under multiprogramming. Hood supports the abstraction of user-level threads, and it schedules those threads onto processes using a

non-blocking implementation of the provably efficient work-stealing algorithm [15]. It employs non-blocking synchronization [30] for the concurrent data structures and judicious use of "yields." We have also studied other alternative implementations that emphasize the importance of the features that Hood has.

We also show that Hood application performance can be bounded by the formula

$$T_P \leq c_1 T_1/P_A + c_\infty T_\infty P/P_A \ ,$$

where c_1 and c_∞ are small constants, and T_∞ is the "critical-path length" of the computation, which, as defined in Chapter 2, is a lower bound on the execution time for any number of processes and processors. This bound holds even when the program runs on a set of processors that grows and shrinks over time, in which case we define P_A as the time-average actual number of processors on which the program runs. Importantly, we find that this bound holds with the constant c_1 very close to 1. Thus, we obtain linear speedup — that is, $T_P \approx T_1/P_A$ — whenever $T_\infty P/P_A$ is small relative to T_1/P_A — that is, whenever P is small relative to T_1/T_∞, a quantity that is naturally interpreted as the "average parallelism" of the computation. We show that this bound holds across all of our applications that were implemented with Hood and across all of the inputs to these applications.

The remainder of this thesis is organized as follows. In Chapter 2 we cover the work-stealing algorithm, we describe the Hood scheduler and we measure and compare the performance of the Hood scheduler and several alternative implementations. Chapter 3 describes the Hood library, the programming interface, the architecture and the implementation. In Chapter 4 we show that the performance of Hood can be modeled with a simple bound based on work and critical-path length and we do further studies based on measurements with multiprogrammed workloads to validate our model. Finally in Chapter 5 we conclude and we consider other proposed solutions to the problem of efficient multithreading in multiprogrammed environ-

ments. Also we discuss some of the limitations of our results and of our approach, and plans for future work to address some of these limitations.

Chapter 2

Hood's work-stealing scheduler

The ***work-stealing algorithm*** dynamically assigns threads to processes for execution in a provably efficient manner [14, 15]. In this chapter, we review the work-stealing algorithm, and we state the proven performance bounds. In addition, we describe the non-blocking implementation of this algorithm [9] that is used in Hood. In the rest of this thesis, we experiment with applications that are coded with Hood. We have also implemented and studied some alternative implementations of the work-stealing algorithm. These alternative implementations perform poorly and reveal the importance of some features of Hood's implementation like the non-blocking deques and the use of yields.

2.1 The work-stealing algorithm

In the work-stealing algorithm, each process maintains its own pool of ready threads from which it obtains work, and when a process finds that its pool is empty, it becomes a thief and steals a thread from the pool of a victim process chosen at random. Each process's pool is maintained as a double-ended queue, or ***deque***, which has a bottom and a top.

To obtain work, a process pops the ready thread from the bottom of its deque and commences executing that thread. The process continues to execute that thread until the thread either blocks or terminates, at which point the process goes back to the bottom of its deque to pop off another thread upon which it can work. During the course of executing a thread, if the thread creates a new thread or unblocks a blocked thread, then the process pushes the newly ready thread onto the bottom of its deque. Thus, so long as a process's deque is not empty, the process manipulates its deque in a LIFO (stack-like) manner.

When a process goes to obtain work by popping a thread off the bottom of its deque, if it finds that its deque is empty, then the process becomes a thief. It picks a victim process at random (using a uniform distribution) and attempts to obtain work by removing the thread at the top of the victim process's deque. If the victim process's deque is empty, then the thief picks another victim process and tries again. The thief repeatedly attempts to steal until it finds a victim whose deque is non-empty, at which point the thief reforms and commences work on the stolen thread as described above. Since steals take place at the top of the victim's deque, stealing operates in a FIFO manner.

2.2 A model of multithreaded computation

This idea of working in a LIFO manner and random stealing in a FIFO manner leads to performance that has been shown, for the case of a dedicated, non-multiprogrammed machine, to be efficient both analytically and empirically. Before stating these results, we first introduce some terminology based on the "dag" structure of multithreaded computations, where we define a ***computation*** as the instructions that are executed when a program is run on an input problem.

We can think of the individual instructions that are executed by the threads in a computation as forming a directed, acyclic graph, or ***dag*** as in Figure 2.1.

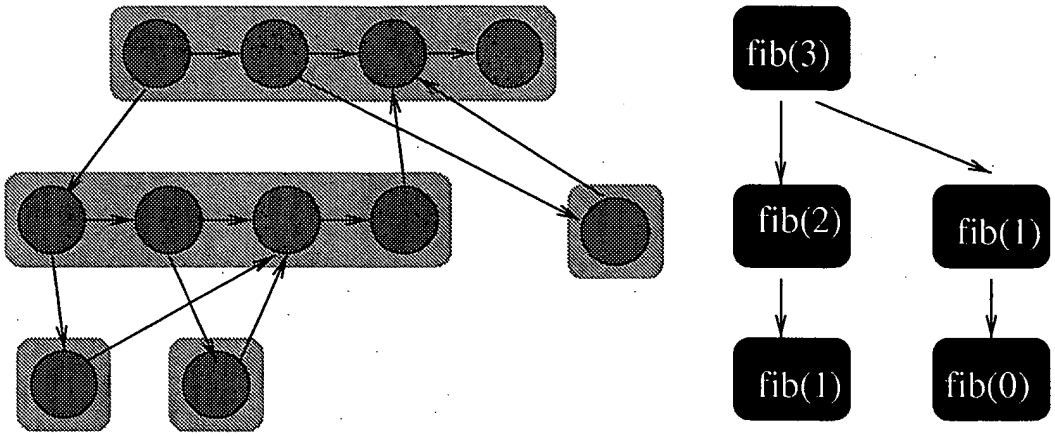

Figure 2.1: A model of multithreaded computation. Threads are shown as rectangular boxes which group a number of instructions. The instructions are shown as circles connected with arrows. Downward arrows indicate "forking" instructions, horizontal arrows show "sequential" execution and upward arrows indicate "unblocking" instructions. This dag corresponds to the computation of the Fibonacci function for the number 3, with the use of threads.

The instructions within any one thread are linked by edges that form a chain according to the thread's dynamic instruction execution order. In addition, when an instruction in one thread creates a new thread, then the dag has an edge from the "forking" instruction in the first thread to the first instruction in the new thread. When the execution of an instruction in one thread unblocks a blocked thread, then the dag has an edge from the "unblocking" instruction in the first thread to the next instruction to be executed in the unblocked thread. In general, whenever threads synchronize such that an instruction in one thread cannot be executed until after some instruction in another thread, then the dag contains an edge from the latter instruction to the former instruction. Using this construction, we view a multithreaded computation as a dag.

We characterize a multithreaded computation with two measures of its dag: work and critical-path length. The ***work***, denoted by T_1, is the sum of the execution

times of all instructions in the dag. Observe that with $P = 1$ process, the process spends all of its time executing threads (it never has to steal), so T_1 is the execution time of the computation with 1 process. The ***critical-path length***, denoted by T_∞, is the maximum sum of execution times for all of the instructions along any (directed) path in the dag. The critical-path length is a lower bound on the execution time for any number of processes and processors. The ratio T_1/T_∞ is called the ***average parallelism.***

Given any multithreaded computation with work T_1 and critical-path length T_∞, we have the following results. The analytical result [15] states that for any number P of processes running on $P_A = P$ dedicated processors, the expected execution time T_P is given by

$$T_P = O(T_1/P + T_\infty) \ . \tag{2.1}$$

Note that because T_1/P and T_∞ are both lower bounds on the achievable execution time (barring any cache effects), this bound states that the execution time is within a constant factor of optimal. The empirical result [14] states that this constant factor is quite small. In particular, the execution time can be bounded tightly according to the formula

$$T_P \leq T_1/P + c_\infty T_\infty \ , \tag{2.2}$$

where c_∞ is a small constant, typically between 1 and 2, that depends on various machine parameters. Thus, we observe linear speedup — that is, $T_P \approx T_1/P$ — whenever T_∞ is small relative to T_1/P — that is, whenever P is small relative to the average parallelism T_1/T_∞.

In Chapter 4, we generalize these results to the case of a non-dedicated, multiprogrammed machine. In other words, we consider the case when we have $P_A < P$. Moreover, we shall allow that the actual number of processors on which the P processes execute can vary over time. We shall, therefore, generalize our definition of P_A to be the "time-averaged" actual number of processors used.

`mm`(n)	Multiply two dense $n \times n$ matrices of doubles using a blocked data layout. Each block is of size 16×16.
`lu`(n)	Compute LU-decomposition without pivoting of a dense $n \times n$ matrix of doubles using a blocked data layout. Each block is of size 16×16.
`barnes`(n, s)	Run Barnes-Hut n-body simulation [12] on n bodies for s time steps. This code is adapted from the SPLASH-2 [52] program, but for the work-stealing version, we parallelized the tree-building with a divide-and-conquer algorithm, so as to avoid the use of locks.
`heat`(n, m, s)	Simulate heat propagation on an $n \times m$ grid for s iterations using Jacobi iteration on a 5-point stencil. This application is very similar to the SPLASH-2 Ocean program [52].
`msort`(n)	Merge sort n integers. Each recursive call is done in parallel, and in addition, the merging is done in parallel using a simple divide-and-conquer technique.
`ray`()	Raytrace scene to compute frame buffer of pixel colors. This application is adapted from the SPLASH-2 [52] program, and we use `balls4.env` as the scene to be rendered.

Table 2.1: Applications used in our study. All applications are written in `C++` and compiled with version 4.1 of the Sun `CC` compiler using flags `-xarch=v8plus -O5 -dalign -noex`. The `mm`, `lu`, `barnes`, and `heat` applications are all easily parallelized with a static partitioning, and our statically partitioned versions are built directly on top of the Solaris `thread` library. Work-stealing versions for all of these applications are built on top of our `Hood` library, which is in turn built on top of the Solaris `thread` library.

2.3 `Hood`'s non-blocking implementation

We now describe `Hood`'s non-blocking implementation of the work-stealing algorithm. This implementation has two key features: the deques, which must support concurrent accesses, are implemented with non-blocking synchronization, and each process, between consecutive steal attempts, performs system calls to "yield" the processor.

In the non-blocking work stealer, the deques are implemented with non-blocking synchronization. That is, instead of using mutual exclusion, we use powerful atomic instructions, notably the SPARC v9 `casxa` [46] (64-bit compare-and-swap) instruction. This instruction operates as follows. It takes three operands: a register with a memory address pointer `addr` and two other registers `old` and `new`.

		T_s	T_1	T_1/T_s	T_∞	T_1/T_∞	T_8	T_1/T_8	T_s/T_8
`mm`(1024)	static	24.78	25.12	1.014			3.28	7.67	7.56
	steal		25.36	1.023	0.01	2536	3.30	7.68	7.51
`lu`(2048)	static	66.85	60.26	0.901			9.41	6.41	7.11
	steal		67.74	1.013	0.05	1394	9.07	7.47	7.37
`barnes`(16384, 10)	static	50.59	52.24	1.033			7.60	6.87	6.65
	steal		52.04	1.029	0.51	102	7.41	7.02	6.83
`heat`(4096, 512, 100)	static	60.15	59.82	0.995			8.01	7.46	7.51
	steal		60.04	0.998	0.23	264	7.93	7.57	7.59
`msort`($32M$)	steal	64.47	61.56	0.955	0.11	540	8.14	7.57	7.93
`ray`()	steal	75.37	77.61	1.030	0.33	235	9.91	7.83	7.61

Table 2.2: Measured application characteristics. For each application, the row labeled "static" (when applicable) represents the statically partitioned version of the application, and the row labeled "steal" represents the work-stealing version of the application, coded using Hood. All times are in seconds. T_s is the execution time of a serial implementation. T_1 is the work of the computation — that is, the execution time with one process. For the work-stealing versions, T_∞ is the critical-path length. T_8 is the execution time with 8 processes running on 8 (dedicated) processors.

It compares the value of the memory address pointed by addr with the value of the old register and if they are the same then it atomically swaps the value of the memory address addr with the value of the new register.

A complete description of this implementation can be found in [9]. This implementation is non-blocking, as opposed to wait-free [30], meaning that it is possible for a process to starve in its attempts to perform steal operations. Livelock, however, cannot occur because if one process starves, then others must be making progress. It turns out that wait-freedom is not needed to prove the analytical result [9] as stated in Section 4.1 — the non-blocking property is sufficient.

In addition to the non-blocking deque implementation, the non-blocking work stealer also makes judicious use of "yields." Each process makes system calls to yield the processor between consecutive steal attempts. We use a combination of `priocntl` (priority control) and `yield` system calls. Whenever a process becomes a thief, it calls `priocntl` to lower its priority. Once the thief has stolen a thread and reformed, it calls `priocntl` to restore its former priority. In addition, when a thief makes an unsuccessful steal attempt, it calls `yield`. In order to mitigate the high cost of these system calls, a thief delays its call to `priocntl` until after it has made enough unsuccessful steal attempts to amortize the cost of the `priocntl` call. Likewise, a thief calls `yield` only after it has made enough unsuccessful steal attempts to amortize the cost of the `yield` call.

To evaluate the non-blocking work stealer empirically, we have coded several applications in C++ on top of Hood. These applications are listed and described in Table 2.1. In addition, Table 2.2 gives a quantitative characterization of each application for a chosen input problem. Figure 1.2 shows the speedup obtained for the non-blocking work stealer. These experiments were run on a dedicated 8-processor machine, so the number P_A of processors used is given by $P_A = \min\{8, P\}$. Here we have achieved our goal. We obtain nearly ideal linear speedup across a

wide range of numbers of processes, even when the number of processes exceeds the number of processors.

Before continuing, we point out that we are defining and measuring speedup as T_1/T_P, as opposed to T_s/T_P, where T_s denotes the execution time for a (good) serial program. We use this definition in order to focus on the performance effects of our scheduler implementation, unencumbered by the overheads induced by other aspects of the implementation. We shall refer to the ratio T_s/T_P as the ***application speedup*** in order to differentiate it from the ***computational speedup*** T_1/T_P, and we note that the two are related by $T_s/T_P = (T_1/T_P)/(T_1/T_s)$. The ratio T_1/T_s measures the ***overhead*** in our parallel implementation. It is the amount of work performed by the parallel computation divided by the amount of work performed by the serial computation. The overhead is affected by many aspects of the implementation that have little to do with the scheduler. The computational speedup is independent of this overhead and measures the scheduler's ability to extract speedup from a computation. For convenience, we use the word "speedup" alone to denote the computational speedup.

From Table 2.2 we observe that the overhead T_1/T_s is near 1.0 for all of our applications. Thus, we are achieving efficient performance not just in a relative sense — that is, relative to T_1 — but in an absolute sense — that is, relative to T_s. For the dedicated case when we have $P_A = P$, our work-stealer is performing just as well as static partitioning, and in the non-dedicated case when we have $P_A < P$, our work-stealer is far outperforming static partitioning. Our work stealer delivers almost perfect linear speedup — computational speedup as well as application speedup — in both cases.

Given the heavy cost of `priocntl` and `yield` system calls, it may come as a bit of a surprise that the non-blocking work stealer produces linear application speedup. An example of the "work-first" design principle [23], the key to this per-

formance is the fact that these system calls occur only when a process is stealing, and this has two important consequences. First, the cost of these system calls does not show up as overhead T_1/T_s, because a 1-process execution never steals and consequently, never performs either of these system calls. Second, we know from prior analytical and empirical work [14, 15] that the number of steals per process grows at most linearly with the critical-path length T_∞ and is independent of the amount of work T_1. Thus, when P is small relative to the average parallelism T_1/T_∞, the execution incurs very few steals, and the cost of these system calls is negligible compared to the work per process. Effectively, the parallelism allows the cost of these system calls to be hidden by the amount of work per process, so linear computational speedup is achieved. The combination of low overhead and linear computational speedup means linear application speedup.

2.4 Alternative implementations of work stealing

In this section we study three alternative implementations of the work-stealing algorithm in order to gain more understanding of the behavior of Hood's non-blocking implementation. These alternative implementations perform poorly and reveal why the non-blocking deques and use of yields are important in practice. The first two implementations use locks (mutual exclusion) as opposed to non-blocking synchronization to implement the deques. Of these two implementations, one uses spinning locks and the other uses blocking locks. The third implementation uses non-blocking synchronization but does not perform yields. We shall refer to this third implementation as "naive non-blocking."

Our two implementations with locks are particularly simple. Each process's deque has a lock associated with it, and each deque operation is surrounded by a lock/unlock pair. Our spinning-lock implementation uses a simple "test-and-set" lock, using a word of memory to represent the lock state and the SPARC v9 casa

(compare-and-swap) instruction to update the state atomically. Our blocking-lock implementation uses the Solaris `thread` library. A lock is a `mutex_t`, and the lock is operated upon with the `mutex_lock` and `mutex_unlock` calls.

Many other and more sophisticated locking strategies are known [4, 25, 31], but we do not consider them. One advantage of some of these strategies is that they perform well under high contention. In our case, each process has its own deque and contention arises only due to thieves, who steal at random. Thus, contention remains constant (and low) even as we add processes or processors. Moreover, as we shall see, the salient problems that we observe for our two locking implementations cannot be fixed with more sophisticated locking strategies. In addition, we shall not consider preemption-safe locking [1, 5, 13, 35, 39], because it requires non-traditional kernel support. In particular, it requires either that the kernel does not preempt processes while they hold locks or that the kernel informs processes of impending preemptions.

Figure 2.2, Figure 2.3 and Figure 2.4, show the measured speedup T_1/T_P plotted against the number P of processes for each of our applications and for each of our three alternative work-stealing implementations. These experiments were run on a dedicated 8-processor machine, so the number P_A of processors used is given by $P_A = \min\{8, P\}$. We observe that all three implementations produce nearly ideal linear speedup for $P \leq 8$ — that is, when $P = P_A$, so we have a dedicated machine. On the other hand, when we have $P > 8$, we have more processes than processors, and we observe that the speedup falls off. This fall off is quite dramatic for the spinning-lock and naive non-blocking implementations, especially for the `heat`, `barnes`, and `lu` applications. In this regard, the blocking-lock implementation is by far the best, as it suffers only a gradual falling off.

Focusing on the scheduler implementation, we shall make quantitative comparisons of (computational) speedup and only make qualitative comparisons of over-

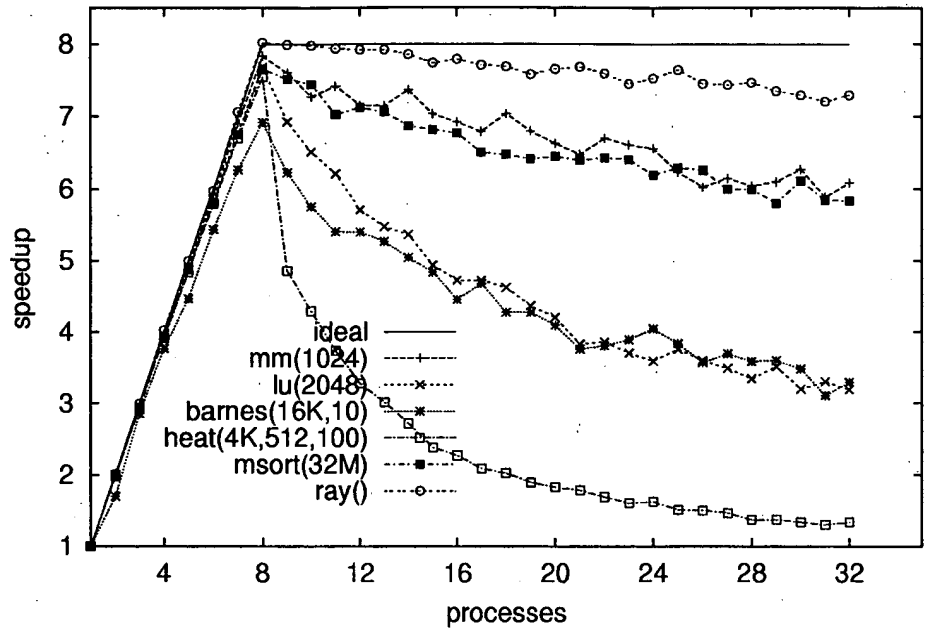

Figure 2.2: The speedup of work-stealing applications plotted as a function of the number P of processes when run on a dedicated 8-processor machine, for an alternative implementation of work stealing with spinning locks.

head. Our implementations are replete with instrumentation, and in order to keep the implementations simple, we have not employed many of the known mechanisms [6, 24, 40] for keeping the overhead low. For this reason, a quantitative comparison of the overhead for our three alternative implementations would be meaningless. We cannot, however, ignore overhead. We shall make qualitative comparisons, and Table 2.2 gives the measured overheads for the case of our (non-naive) non-blocking implementation. All of our applications perform a reasonable amount of work between synchronizations, and the overhead for this implementation is very near 1.0 in all cases.

Returning to the measured speedups for our three alternative implementations, we first consider the spinning-lock implementation. This implementation is

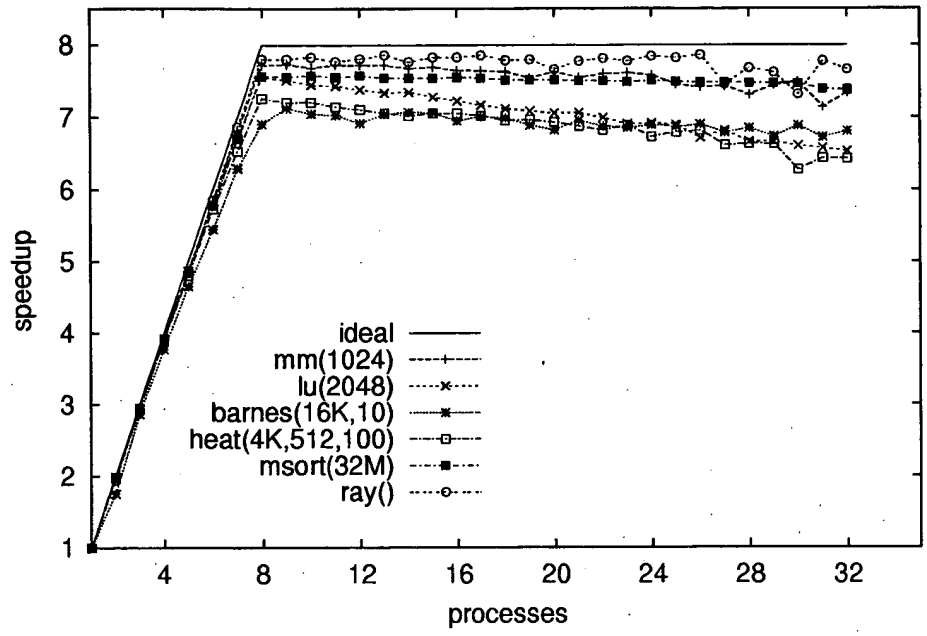

Figure 2.3: The speedup of work-stealing applications for an implementation of work stealing with blocking locks.

often used in practice, because it has very low overhead. Locking and unlocking takes a small handful of user-level instructions. Unfortunately, as we see in Figure 2.2, this implementation performs poorly when the number P of processes exceeds the number of processors. In Figure 2.5 we break down the execution time for three of the applications that perform particularly poorly. We consider the cases $P = 8$ and $P = 16$. The bars show clearly that in going from 8 to 16 processes, the time spent trying to acquire locks goes up dramatically. If a process acquires a lock and then gets preempted by the kernel scheduler, then when other processes go to acquire that lock, they will simply spin until the holding process gets to run and release the lock. This type of behavior will be seen in any implementation of spinning locks, no matter how clever it is in dealing with contention.

This problem is well known [27] and traditionally is fixed by using blocking

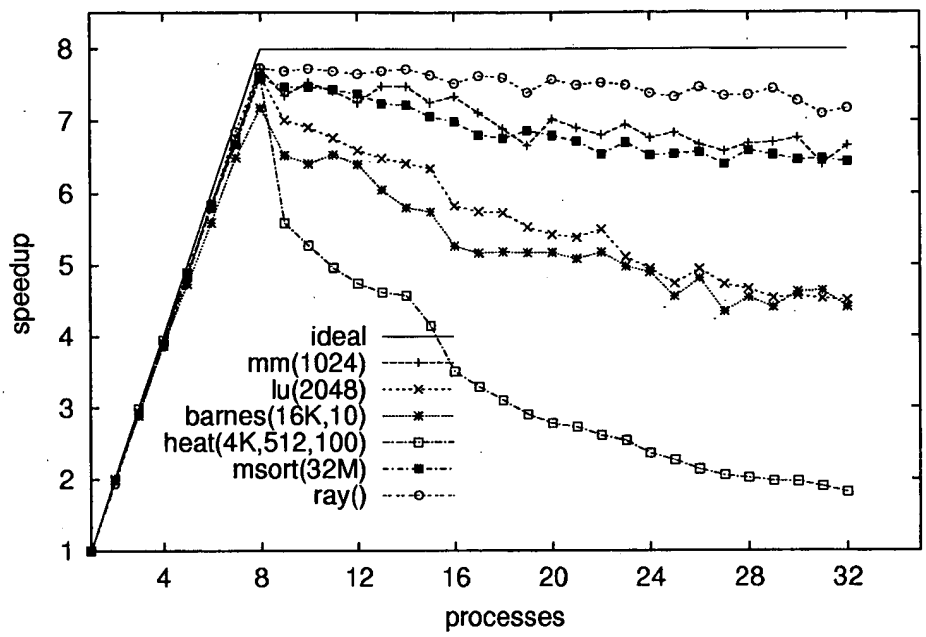

Figure 2.4: The speedup of work-stealing applications for a naive non-blocking implementation of work stealing.

locks, and as we see in Figure 2.3, the blocking-lock implementation performs much better. In this case, if a process acquires a lock and then gets preempted by the kernel scheduler, then when other processes go to acquire that lock, they will be put to sleep by the kernel scheduler, thereby freeing up their processors so that the holding process can run and release the lock. Of course, the resulting frequent context switches may degrade performance, and we do indeed see some performance degradation as the number of processes grows beyond the number of processors. In Figure 2.6 we see that as we go from 8 to 16 processes, the amount of time spent working goes up slightly. This extra time is due to cache misses resulting from the frequent context switches, though this effect is mitigated by the Solaris 2.5.1 affinity scheduler. We counted (user-process to user-process) context switches using the Solaris kernel's TNF probes. For the `lu` application at $P = 8$ we counted 145

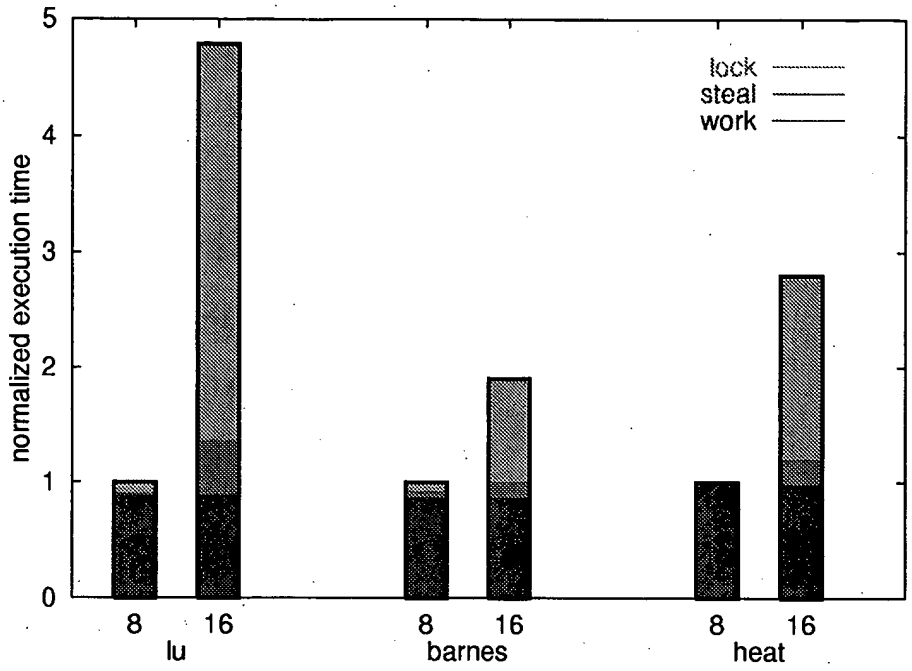

Figure 2.5: A breakdown of the execution time of three of the work-stealing applications at $P = 8$ and $P = 16$, for an implementation of work stealing with spinning locks. The bars at $P = 8$ have all been normalized to an execution time of 1.0, and the bars at $P = 16$ have been scaled accordingly. The bottom (dark) section of each bar is the time spent executing threads. The middle (medium gray) section is the time spent stealing threads. The top (light gray) section is the time spent trying to acquire locks.

context switches, and at $P = 16$ we counted 8536 context switches. In contrast, with our spinning-lock implementation, at $P = 8$ we counted 0 context switches, and at $P = 16$ we counted 616 context switches. Similar numbers were counted for the other applications.

In addition to the frequent context switches, blocking locks have a very high overhead. Locking and unlocking requires calls into the kernel that are expensive, and this overhead is only expected to get worse for the foreseeable future [7]. This overhead will be present in any implementation of locks that admit blocking in the

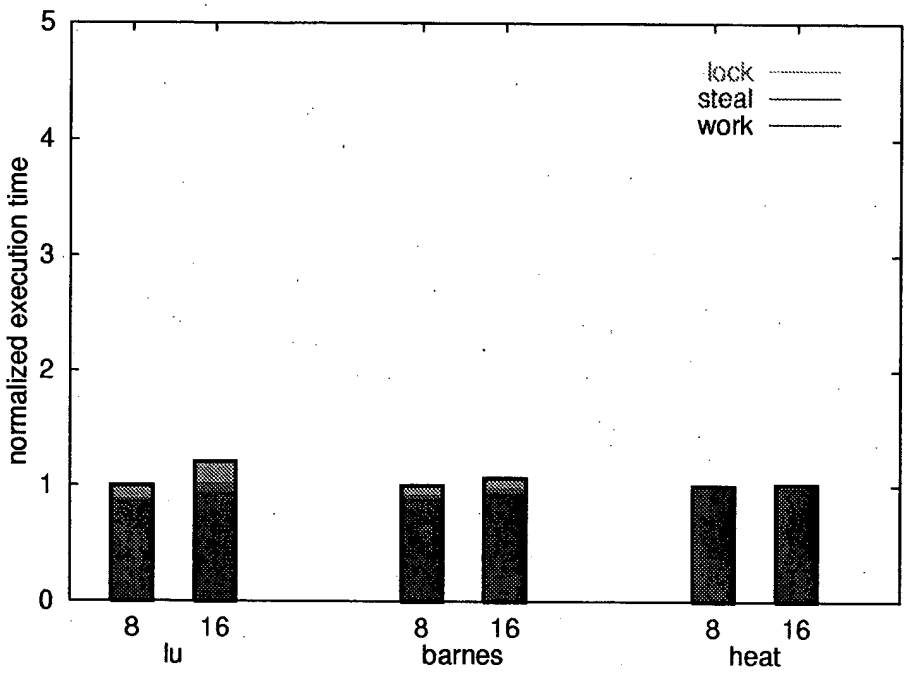

Figure 2.6: A breakdown of the execution time of three of the work-stealing applications at $P = 8$ and $P = 16$, for an implementation of work stealing with blocking locks.

kernel. A hybrid spin-then-block lock can reduce the number of context switches [19], but it still requires kernel support for blocking. In contrast to the `priocntl` and `yield` system calls in the (non-naive) non-blocking implementation, the use of blocking locks violates the work-first design principle [23], and the overhead cannot be hidden. In a user-level thread scheduler, we do not want to go into the kernel every time we schedule a thread.

The naive non-blocking implementation was supposed to fix the problems with spinning locks and blocking locks, but as we see in Figure 2.4, it did not. Nevertheless, the naive non-blocking implementation does exhibit some good properties. It never wastes time spinning on a lock, and the implementation is done at

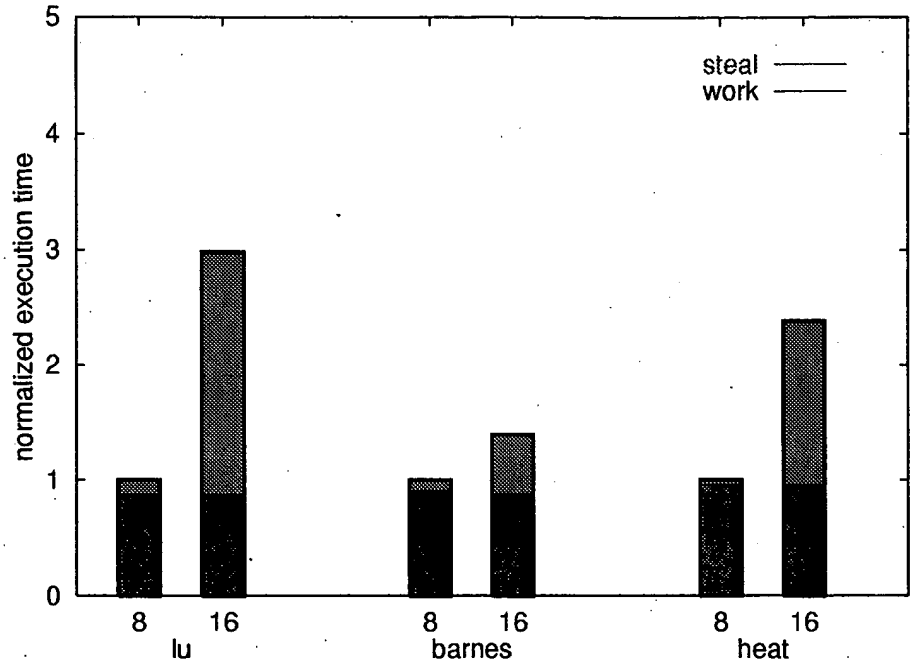

Figure 2.7: A breakdown of the execution time of three of the work-stealing applications at $P = 8$ and $P = 16$, for a naive non-blocking implementation of work stealing.

user level with low overhead. Moreover, we count very few context switches. For the lu application at $P = 8$ we counted 3 context switches, and at $P = 16$ we counted 336 context switches. Similar numbers were counted for the other applications. So why do we observe poor performance? In Figure 2.7 we see that as we go from 8 to 16 processes, the amount of time spent stealing goes up dramatically. This time increase comes from a huge increase in the number of steal attempts, and from program traces with TNF probes, we find that these steal attempts occur in bursts.

The activity that causes these bursts is as follows. A process that is in the middle of executing a thread gets preempted by the kernel scheduler. The other processes that are running continue to execute threads from their deques, and when

their deques become empty, they steal threads, possibly from the deque of the preempted process. Eventually, we get to a point where every deque is empty. The only runnable thread is the one that is being executed by the preempted process. All other threads are waiting on this thread for some type of synchronization. Thus, the processes that are running continue to make failed steal attempts, because there is nothing that can be stolen. These processes spin trying to steal until eventually the preempted process gets to run and continue executing its thread, which forks or unblocks other threads. Note that we do not see this behavior with the blocking-lock implementation, because a process never gets to run long enough that it might get preempted in the middle of executing a thread. We see a little bit of this behavior with the spinning-lock implementation, but in this case, processes spend far more time spinning on locks than spinning trying to steal.

To prevent this behavior, Hood's non-blocking implementation uses `priocntl` and `yield` system calls. A process that is spinning trying to steal will be running at low priority and making repeated calls to `yield`. Thus, such a process will relinquish its processor, thereby allowing the preempted process to run. We shall consider the three alternative implementations no further and we will only refer to the non-blocking implementation of Hood.

Chapter 3

The Hood library

Hood is a C++ user-level threads library targeted for shared-memory multiprocessors. It supports the abstraction of user-level threads and it schedules those threads onto processes using the work-stealing algorithm. The current implementation of Hood supports blocking, non-preemptive threads. A blocking thread can create child threads and then join with them waiting for the children to return. Threads can use synchronization variables such as locks and semaphores.

The Hood runtime system has three main entities: threads, serial machines and parallel machines. Each one of them is represented with a C++ class. A parallel machine can be thought as a "virtual multiprocessor" that creates a number of "virtual processors", the serial machines. Each serial machine has a local heap where it allocates and deallocates space for the threads and a deque, where the threads are pushed when they are ready for execution. The serial machine executes the work-stealing algorithm using the non-blocking implementation that we described in Section 2.3. It pops a thread from the deque and executes it or if the deque is empty it tries to steal from the deque of another serial machine.

Hood has been instrumented to measure work and critical path length. The work is measured by adding up the elapsed time over each thread dispatch. Critical-

path length is measured by timestamping [14].

In this chapter we give a detailed presentation of the Hood library. Section 3.1 is an introduction on how to program parallel applications using the Hood library. In Section 3.2 we describe in more depth the interface classes of the library giving some details about their implementation. Finally in Section 3.3 we present the architecture and some lower-level details about the implementation of the library.

3.1 Programming with Hood

In this section we describe how to program parallel applications with the use of the Hood library. We will start with a simple program that introduces the programming interface that Hood provides. The next section provides more specific information about the library's classes.

We will write a simple program that computes the Fibonacci function recursively, using the Hood library. In C++, a serial version of this program can be written as shown in Figure 3.1. Although this version is not very efficient, it is easily parallelizable.

In comparison Figure 3.2 shows how to write a similar but parallel program that computes the Fibonacci function using Hood. A parallel program that does the same computation as the serial one should be able to execute the two recursive calls in procedure `Fib` in parallel and then block until both of these procedures return, so that it can sum their results and return the final result. The procedure is replaced in the parallel version of the program with a Hood thread. The thread creates two child threads that can be executed in parallel and then waits for the children to finish. The code that each thread executes is placed in the `run` method of the thread. The `new` operator creates the child thread and the `post` method pushes the child thread in the pool of threads that are ready for execution. After the `post` call the parent continues execution, but now the child thread can be executed

```
#include <iostream.h>
#include <stdlib.h>

int  Fib (int n)
{
  // Check base case.
  if ( n < 2 )
    return n;
  // Recursive case.
  else {
    int x, y;

    x = Fib (n-1);
    y = Fib (n-2);

    return (x+y);
  }
}


int main (int argc, char* argv[])
{
  int n, result;

  n = atoi(argv[1]);
  result = Fib (n);

  cout << "Fib(" << n << ") = " << result << ".\n";
  return 0;
}
```

Figure 3.1: A serial C++ program to compute the nth Fibonacci number.

in parallel with the parent in another serial machine if a steal occurs. The serial machines are the "virtual processors" that schedule and execute the threads using our implementation of the work-stealing algorithm. The `waitChildren` call forces the thread to wait until a specific number of its children finish. The children use the `signal` call to notify the parent thread by decrementing the number of signals that the parent is waiting for. When the number of signals that a thread is waiting for becomes zero the thread is ready for execution. In each thread we pass the Fibonacci value to compute, a pointer to the place where the result is going to be

stored and a pointer to the parent thread.

In Figure 2.1 we can see the computation dag for the Fibonacci function with the number 3 as input. The downward arrows start from the `post` instruction of the parent thread and end at the first instruction of the child thread. The upward arrows start from the `signal` instruction of the child thread and end at the `waitChildren` instruction of the parent thread.

At the beginning of the main program we create the parallel machine that will schedule the execution of the threads. The parallel machine is the "virtual multiprocessor" and an object of the `HoodParMach` class. The constructor for `HoodParMach` takes an optional argument that is the number P of "virtual processors" to be used. The default value for this argument is the number of processors in the machine. The constructor creates P instances of the `HoodSerMach` that are ready to execute the threads.

After the `HoodParMach` object is created we post the initial thread in the first `HoodSerMach` and then call the `runScheduler` method of the `HoodParMach` to start the execution. The last thread to be executed is the `FinalHoodThread` which is a special thread that causes the termination of the parallel machine.

`Hood` has been instrumented to measure work and critical-path length. The work is measured by adding up the elapsed time over each `run` invocation. Critical-path length is measured by timestamping all `post` and `signal` invocations [14]. After calling `runScheduler`, an application can call other `HoodParMach` methods to return the elapsed time, the work, or the critical-path length. Many other statistics are also collected and made available.

3.2 Hood's interface classes.

The `Hood` library has three interface classes that the programmers can use, the `HoodThread`, the `HoodSerMach` and the `HoodParMach` class. In order to use the

```
#include "hood.h"
#include <iostream.h>
#include <stdlib.h>

class FibThread : public HoodThread {
private:

  HoodThread* parent;  // Parent thread.
  int* result;         // Location to store result.
  int n;               // Value to compute fib of.

public:

  // Constructor.
  FibThread (HoodThread* parent_, int* result_, int n_)
    : parent (parent_), result (result_), n (n_) {}

  // Thread.
  void run (HoodSerMach* mach);
};

void FibThread::run (HoodSerMach* mach)
{
  // Check base case.
  if ( n < 2 ) {
    *result = n;
    mach->signal (parent);
  }
  // Recursive case.
  else {

    int x, y;
    mach->post (new(mach) FibThread (this, &x, n - 1));
    mach->post (new(mach) FibThread (this, &y, n - 2));

    mach = mach->waitChildren(this,2);
    *result = x + y;

    mach->signal (parent);
  }
}


int main (int argc, char* argv[])
{
  int n, result;

  n = atoi(argv[1]);

  HoodParMach parMach;
  FinalHoodThread* ft =
    new(parMach[0]) FinalHoodThread();
  parMach[0]->post (new(parMach[0])
                    FibThread (ft, &result, n));

  parMach.runScheduler();

  cout << "Fib(" << n << ") = " << result << ".\n";

  return 0;
}
```

Figure 3.2: A Hood program to compute the nth Fibonacci number.

library interface the programmer has to include the `hood.h` header and link the program with the Hood and the Solaris thread libraries.

The `HoodThread` class represents the user-level threads. A program creates threads by instantiating objects of any class that is a subclass of the `HoodThread` class. All threads contain an integer instance variable `joinCount` and a virtual method `run`. The `joinCount` is the number of signals that a thread is waiting for. If it is zero the thread is ready for execution. The `run` method contains the actual code that is executed when a process is running the thread. Each process repeatedly pops a thread off the bottom of its deque and calls the thread's `run` method .

A thread can be in one of the following states: dead, alive, ready, running, or blocked. Figure 3.3 shows the different states of a thread. The arrows represent the transitions that can happen. The solid lines represent the actions that the programmer can do while the dotted lines represent actions that the system does internally. Initially the `new` operator allocates memory and calls the constructor of the thread, transforming its state from dead to alive. An initial value for the `joinCount` is given as an argument to the thread's constructor. If a thread is created with a `joinCount` equal to 0 then the thread is actually ready for execution but is still in the alive state. The `post` method can be used to actually change the thread's state to ready. If the initial value of the `joinCount` is greater than 0, then the thread can become ready with the use of the `signal` method. A thread that is ready is waiting to be executed in the deque of a serial machine. The serial machine always pops a thread from the bottom of the deque and executes it. If the thread has moved from the alive to the ready state then the serial machine will start executing the thread's `run` method. Otherwise if the ready thread was previously in the blocked state, it will restore the thread's context and continue its execution. At this point a thread is in a running state. A running thread can block and wait for a number of threads to "signal". This can be done with the `waitChildren` and `wait`

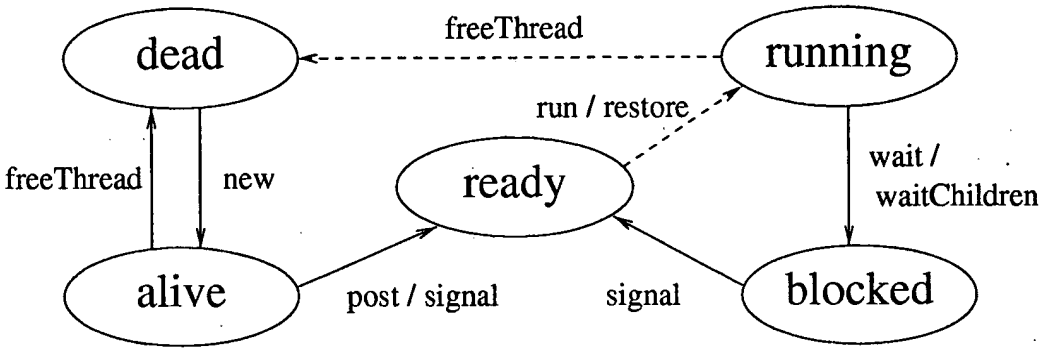

Figure 3.3: States of a Hood thread

methods. In the first method the thread continues executing its children and only blocks if a child is stolen, but with the `wait` call the thread blocks immediately. In both cases the thread is in a block state. After the execution of the `run` method is over the thread is dead and the serial machine uses the `freeThread` method to delete it. A dead thread can than be reused. The programmer should be careful not to delete the threads that are going to be executed from the serial machine. It should only delete threads that are not used from the serial machine using the `freeThread` method of the serial machine.

The `HoodSerMach` is associated with each process. It contains the process's deque and it supports four public methods that threads may use: `post`, `signal`, `waitChildren`, `wait`. The programmer can subclass the `HoodSerMach` if there is a need for data that are specific for each serial machine. When a process executes a thread by calling a thread's `run` method, the process passes, as an argument to `run`, a pointer to its `HoodSerMach`. The programmer uses this pointer to invoke the serial machine methods that change the state of the thread. When a thread creates a thread with a `joinCount` of zero, it can then call the `HoodSerMach`'s `post` method with a pointer to that thread. This method pushes the thread onto the bottom of the `HoodSerMach`'s deque. In order for a thread to signal another thread (whose

joinCount is non-zero), the signaling thread calls the HoodSerMach's signal method with a pointer to the thread being signaled. This method decrements the thread's joinCount, and if the joinCount becomes 0, then it pushes the thread onto the bottom of the HoodSerMach's deque. The waitChildren and wait methods cause a thread to wait for the signals of other threads before they continue execution. Both of these methods return a pointer to the serial machine that resumes the blocked thread. The programmer must be careful to use that updated pointer of the serial machine.

A HoodParMach represents the Hood runtime system. The constructor for HoodParMach takes an optional argument that is the number P of processes to be used. A second optional argument defines the size for the stack that a process uses when it executes a thread. The constructor creates P instances of the HoodSerMach class and it forks P processes (bound threads), giving each process a pointer to its HoodSerMach. When the processes begin executing, they simply park on a condition variable. All of the action takes place in the HoodParMach's runScheduler method. This method releases the P processes and then waits until all P processes have reparked themselves. After being released, each process begins executing its scheduling loop: executing threads from its deque and stealing when its deque is empty. The scheduling loop terminates when a special FinalHoodThread is executed, at which point the P processes repark so the HoodParMach's runScheduler method can return.

A typical application will create a "root" thread and a HoodParMach object. It will post that root thread to any one of the HoodParMach's HoodSerMach objects (which can be accessed by the HoodParMach's subscripting operator), and then it will call the HoodParMach's runScheduler method.

The programmer has also the option to use synchronization objects like semaphores and mutex locks. The semaphores are represented with the HoodSem class.

```
#include "hood.h"

class Buffer {
  int* buff;
  int size;
  HoodSem* full;
  HoodSem* empty;
  int nextin;
  int nextout;
  HoodSem* plock;
  HoodSem* clock;
public:

  Buffer(int size_): size(size_) {
    buff = new int[size];
    empty = new HoodSem(size); full = new HoodSem(0);
    plock = new HoodSem(1); clock = new HoodSem(1);
    nextin = nextout = 0;
  }

  ~Buffer() {
    delete buff; delete empty; delete full;
    delete plock; delete clock;
  }

  HoodSerMach* produce(HoodThread* prodt, int value);

  HoodSerMach* consume(HoodThread* cont, int *value);
};
```

```
HoodSerMach* Buffer::produce (HoodThread* prodt, int value)
{
  HoodSerMach* serMach;
  serMach = empty->wait(prodt);
  serMach = plock->wait(prodt);

  buff[nextin] = value;
  nextin++;
  nextin %= size;

  plock->signal(prodt);
  full->signal(prodt);
  return serMach;
}

HoodSerMach* Buffer::consume (HoodThread* cont, int *value)
{
  HoodSerMach* serMach;
  serMach = full->wait(cont);
  serMach = clock->wait(cont);

  *value = buff[nextout];
  nextout++;
  nextout %= size;

  clock->signal(cont);
  empty->signal(cont);
  return serMach;
}
```

Figure 3.4: A Producer/Cosumer example using the HoodSem class.

The constructor of the class takes as an argument the initial value for the counter of the semaphore. A thread can wait on a semaphore with the `wait` method or it can atomically increment the counter of the semaphore with the `signal` method of the semaphore. The `wait` method blocks the thread until the counter of the semaphore becomes greater than zero and then it atomically decrements it. The `tryWait` method tries to atomically decrement the counter of the semaphore when the counter is greater than zero, like the `wait` method and returns immediately even if the attempt fails. The return value indicates if the attempt was succesfull or not. The programmer has also the option to use mutex locks for synchronization. The mutex locks are represented with the `HoodLock` class. The class has two methods `lock` and `unlock` for locking and unlocking the mutex. The `tryWait` method can also be used with mutexes, if the programmer wants to try to lock the mutex, but does not want to block if the attempt fails. Both the `wait` and the `lock` method return a pointer to the serial machine that resumes the blocked thread, if the thread had to block. The programmer must be careful to use that updated pointer of the serial machine.

Figure 3.4 shows a `Hood` implemantation of the well-known producer/consumer problem using the `HoodSem` class. We have a finite-size buffer and two classes of threads *producers* and *consumers*. The *producers* add items in the buffer with the `produce` method and the *consumers* remove items from the buffer with the `consume` method. Our implementation uses 4 semaphores to ensure proper use of the buffer. Two of them, `full` and `empty` represent the number of full and empty slots in the buffer. They ensure that *producers* wait until there are empty slots in the buffer and that *consumers* wait until there are full slots in the buffer. The other set of semaphores, `plock` and `clock` controlls the concurrent access to the buffer slots from multiple producers and multiple consumers. Both the `produce` and `consume` methods return a pointer to the serial machine. This in neccesssary because after

we return from this methods the serial machine might be different, because of their use of the `wait` method.

3.3 The Hood implementation

Our implementation of Hood is built on top of the Solaris `thread` library and it implements each process as a Solaris Light-Weight Process (LWP). Each LWP can be thought as a virtual processor that is available for executing code or system calls. Our architecture has two scheduling levels: our work-stealing scheduler runs at user-level and schedules threads onto a fixed collection of processes, while below the operating-system kernel schedules processes onto a fixed collection of processors. In Figure 3.5 we can see all the important entities of the architecture. The parallel machine of Hood is a collection of Solaris LWPs. The LWPs are scheduled with the solaris kernel scheduler that assigns the LWPs to the existing processors of the system. Each LWP is associated with a serial machine. The core of each serial machine is our non-blocking work-stealing scheduler. The schedulers execute the Hood user-level threads using the work-stealing algorithm for scheduling. The user-level threads are instances of the `HoodThread` class and they should not be confused with the user level or unbound threads of the Solaris `thread` library. Those threads are not used in the implementation of our library and also they can not be executed from our scheduler.

One difference between the two levels of scheduling is that the Hood schedulers do non-preemptive scheduling while the kernel schedules the LWPs with preemption. The LWPs are scheduled by the kernel onto the available CPU resources according to their scheduling class and priority [18]. There are no priorities in the `HoodThread` class. Thread specific data can be implemented as subclasses of the `HoodThread` class. Using the same method if we subclass the `HoodSerMach` class we can have specific data for each serial machine.

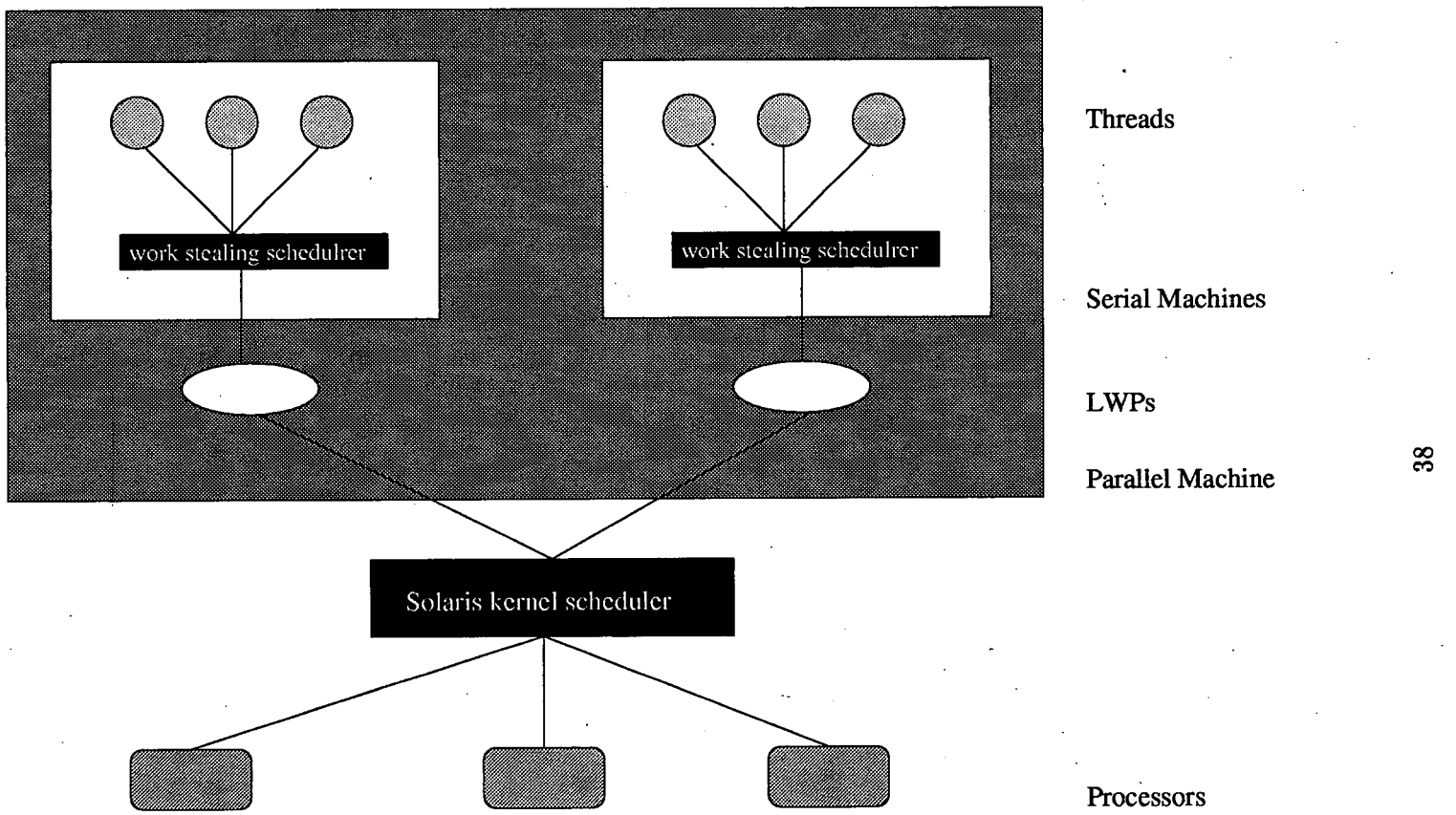

Figure 3.5: The architecture of the Hood library.

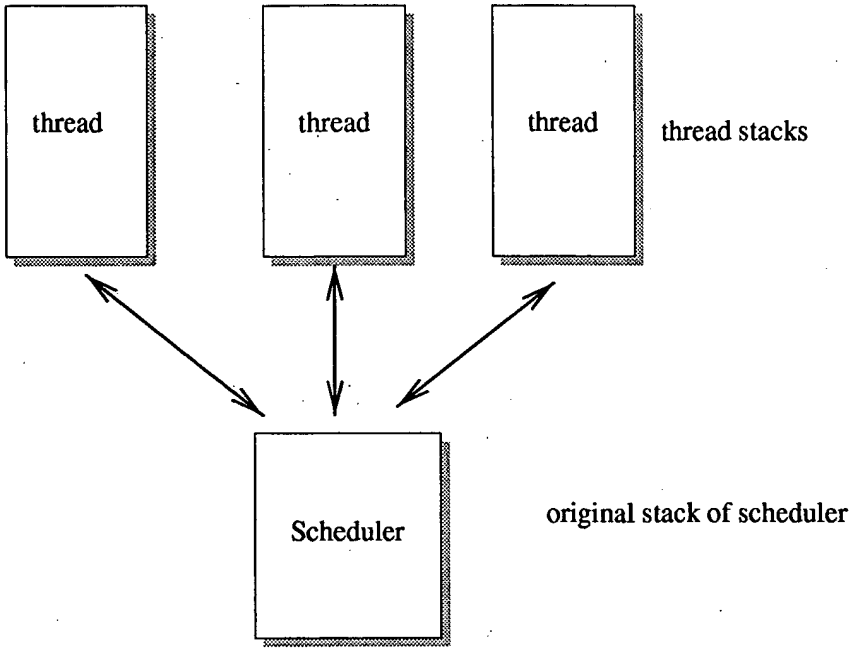

Figure 3.6: Stack implementation in Hood.

In our implementation each thread has a register state and a stack. Whenever the scheduler dispatches a new thread it also allocates a new stack for it. If a thread blocks for a synchronization variable or because it is waiting for signals from other threads we context-switch and return control to the scheduler that continues to execute the work-stealing algorithm. Although it seems like there is a big overhead because of the context-switching this is not actually true. Whenever a thread is waiting for signals from other threads, the scheduler can continue executing the child threads in the same stack with the parent thread until the thread receives all the signals that it is waiting for. With this method that follows the "work-first" principle [23] we avoid the overhead of the context-switch most of the time. A context-switch will happen only if a child of the parent thread is stolen and the serial machine of the parent thread executes all the other children, leaving the deque

empty. In that case the serial machine has to steal, so we need to switch back to the original stack of the serial machine. Each serial machine has an original stack that the threads use to switch to when they need to block. All context-switches happen between a thread and a serial machine. There is no direct context-switching between threads.

3.3.1 Context-Switching

There are two main types of context-switch. The first occurs when a serial machine switches from the original stack to a thread's stack. This can happen when the serial machine dispatches a new thread for execution, in which case it allocates and assigns a new stack to the thread and it moves to the thread's new stack to start the execution of the thread. It also happens when it resumes the execution of a thread that was previously blocked and returns control to the stack of the thread. The second type occurs when the thread blocks and returns control to the original stack of the serial machine that is executing the thread. When the thread resumes, the new serial machine might be different then the one that was executing the thread when it blocked.

We have implemented three different routines that we use to implement the context-switch. All of them are leaf procedures written in the assembly language of the SPARC v9 architecture [46]. The first of them `hoodSave` is used to save the current state of the thread or the serial machine. But a return from `hoodSave` can also occur when the state of a thread is restored. We use the return value of `hoodSave` to separate the two cases. A return value of 0 indicates that the procedure was normally executed. All other values indicate that we return from a restore. The procedure `hoodRestore` restores the state of the thread that was saved with the `hoodSave` call. It takes an extra parameter that is the return value after we restore the state of the thread. As we already mentioned this value should never be

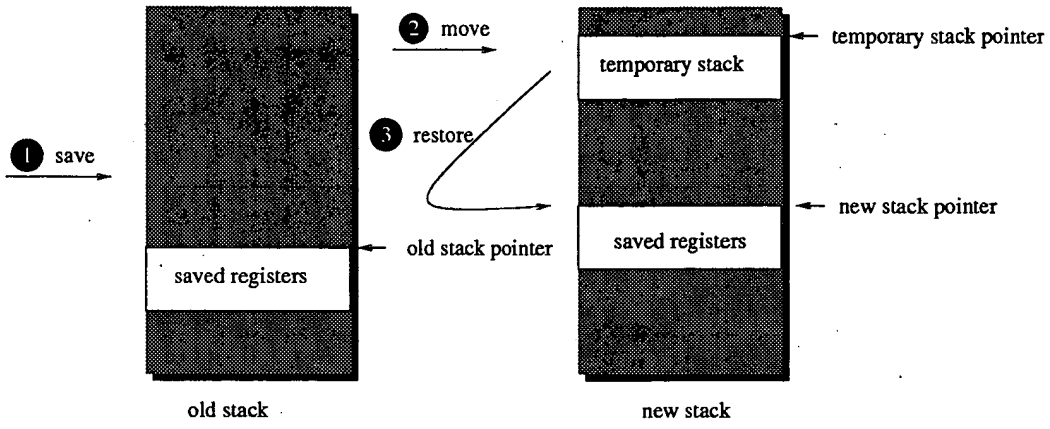

Figure 3.7: The three phases of a context-switch in Hood.

0. The third procedure that we use is `hoodMove` that changes only the current stack pointer. The `hoodMove` was implemented because sometimes whe need to preswitch. That is we save the state of the old thread but before we block, we move to another stack and there we execute some code of the old thread using our new stack and then we continue on the new stack or restore a state in the new stack. This is necessary to avoid race conditions. Since a thread can be blocked in one serial machine and resumed in another there might be a race of the two serial machines for the same stack. So we must make sure that it is never the case that two serial machines use the same stack. Figure 3.7 shows the three different phases of a context-switch.

The state of a thread in our implementation consists of a number of SPARC registers. The SPARC architecture [46] has four groups of integer registers. Each group has eight registers numbered from 0 to 7. These are the *in*, the *local*, the *out* and the *global* registers. Whenever we save the state of a thread we write all the *in* and the *local* registers and the `%o7` register from the *out* registers in the stack. The registers `%i0` to `%i5` may contain input parameters for the function that calls the `hoodSave` function. Register `%i6` is the current frame pointer and `%i7` is the

return address for the parent function of `hoodSave`. Also we need to save register %o7 which contains the return address of the save function and register %o6 which is the current stack pointer. All the registers are saved in the the stack except the current stack pointer which is saved in the thread structure and is used as a pointer to retrieve the rest of the registers when we restore the state of the thread.

The SPARC architecture uses register windows [32] that have sets of *in* and *local* registers. The *out* registers of a set are used as the *in* registers for the next set that belongs to the next function call. Each set can be spilled on the stack whenever there is overflow on the window, because there are not enough clean register sets. The save area is the current top of the stack, so the stack pointer must always point to the correct place. When a serial machine does a save or a restore we always need to flush all the registers from the register window to the stack area and make sure that the effects of all the stores are visible by the other processors before any loads are performed.

3.3.2 Non-blocking synchronization

As we have already mentioned, the threads can synchronize with the use of the `joinCount` variable. Whenever a thread signals another thread it decrements its join counter. Since a lot of threads can try to decrement a thread's join counter at the same time it is obvious that we have a race condition. But instead of using mutual exclusion, this decrement is implemented with a simple non-blocking method that uses the powerful atomic instruction `casxa` (64-bit compare-and-swap), of the SPARC v9 architecture [46]. This instruction is used in other places that we need non-blocking synchronization like the deques and the semaphores.

The threads can also synchronize with the use of synchronization objects like the semaphores and the locks. The `HoodSem` class consists of a counter and a pointer to a list of threads that are blocked because of a `wait` call, waiting to access the

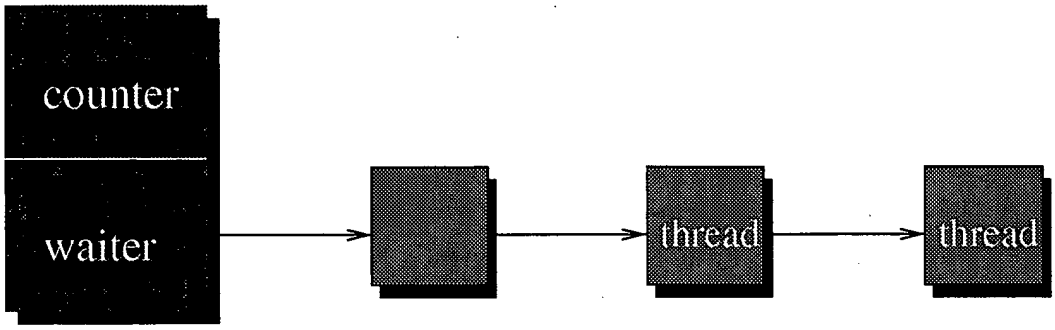

Figure 3.8: A `HoodSem` object. It contains a counter and a pointer to a LIFO list of threads that are blocked waiting for the semaphore.

semaphore. In order to imlement the semaphore in a non-blocking manner using `casxa`, we packed the semaphore structure into a single 64-bit word that can be operated on atomically with `casx`. The LIFO waiting list is implemented using the `next` field of the `HoodThread` class. This field is updated whenever a thread is added or removed from the waiting list of a semaphore. The class has three methods `tryWait`, `wait` and `signal` that we described in Section 3.2. All the three methods employ simple load-modify-compare&swap loops that try to atomically change the semaphore structure. Figure 3.9 presents some more details about the implementation of this methods. The `HoodLock` class is internally implemented as a binary semaphore.

```
signal(){
  do {
    if (!waiter)
      increment counter;
    else {
      remove waiter from list;
      put removed waiter in ready deque of scheduler;
    }
  } until success;
}
```

```
trywait(){
  do {
    if (counter>0)
      decrement counter;
  } until success OR
    loop executed  SPIN times;
}
```

```
wait(){
  if (tryWait()) return;
  else {
   save state;
   move to scheduler's stack;
   call add();
  }
}
```

```
add(){
  do {
    if (counter>0){
      decrement counter;
      resume thread in wait() call;
    }
    else {
      add thread to list;
      block thread and switch to scheduler;
    }
  } until success;
}
```

Figure 3.9: The HoodSem class methods. The operations increment/decrement counter and add/remove from list are executed atomically and always modify the HoodSem structure as one 64-bit word.

Chapter 4

The performance of HOOD

In this chapter we generalize the model that was presented in Section 2.2 to the case of non-dedicated, multiprogrammed environments. In other words, we consider the case when we have $P_A < P$ where P_A is the number of processors and P is the number of processes. Moreover, we allow that the actual number of processors on which the P processes execute can vary over time. We therefore, generalize our definition of P_A to be the "time-averaged" actual number of processors used. The performance of Hood can be modeled with a simple bound based on work and critical-path length. We validate our model with studies on dedicated machines and then do further studies based on measurements with multiprogrammed workloads. The studies verify our model and show that Hood delivers efficient performance under multiprogramming. For all the Hood prototype applications, we observe linear speedup whenever the number of processes is small relative to the average parallelism.

4.1 Performance modeling

Hood admits a simple performance model based on work and critical-path length that has been proven analytically, and in this section we provide empirical evidence

as to the validity of this model. The analytical result [9] states that for any number P of processes, the execution time T_P is given by

$$T_P = O(T_1/P_A + T_\infty P/P_A) \ , \tag{4.1}$$

where T_1 is the work of the computation, T_∞ is the critical-path length of the computation, and P_A is the time-average number of processors that actually execute the computation. This analysis treats the kernel scheduler as an adversary with the one provision that it must obey yields.

To quantify this relationship, we replace the big-Oh notation with explicit constants. According to the asymptotic bound, there exist constants, c_1 and c_∞, such that the execution time is bounded by

$$T_P \leq c_1 T_1/P_A + c_\infty T_\infty P/P_A \ . \tag{4.2}$$

In this section, we show that this bound holds with very small constants, c_1 and c_∞. Most importantly, we find that the constant c_1 is very close to 1. Thus, we observe linear speedup — that is, $T_P \approx T_1/P_A$ — whenever $T_\infty P/P_A$ is small relative to T_1/P_A — that is, whenever P is small relative to the average parallelism T_1/T_∞. Note that in the case of a dedicated, non-multiprogrammed machine, we have $P_A = P$, and the model of Inequality (4.2) is identical to the previous model of Inequality (2.2). We show that this bound holds across all of our Hood applications and across all of the inputs to these applications.

Our bound, Inequality (4.2), has four independent variables — T_1, T_∞, P, and P_A — and we would like to show that this bound holds across all values of all of these variables. It turns out that we can perform a straightforward algebraic manipulation to derive a simpler bound that aggregates some of these variables. Recall that we define the utilization as $T_1/(P_A T_P)$. If we plug Inequality (4.2) into this definition and divide the top and bottom by T_1, then we obtain the following

bound for utilization:

$$\frac{T_1}{P_A T_P} \geq \frac{1}{1 + cP/(T_1/T_\infty)} \,. \tag{4.3}$$

Notice that the utilization is lower-bounded by a function of only one independent variable, $P/(T_1/T_\infty)$, that we call the ***normalized number of processes***. In other words, if our model is accurate, then we should be able to lower-bound utilization as a function of the normalized number of processes according to Inequality (4.3). Observe that this model says that when the normalized number of processes is much less than 1.0 — that is, when we have $P \ll T_1/T_\infty$ — then the utilization should be near 1.0. As the normalized number of process gets large relative to 1.0, the utilization may drop off.

We now wish to validate our model by running our applications with different input problems that generate different values of T_1 and T_∞ and with different numbers P of processes to see if Inequality (4.3) holds. Somewhat limiting our ability to perform such an experiment, we find that for all of our applications, the input problems that generate reasonable values of T_1 tend to generate values of T_∞ within a very narrow range. Thus, we shall begin our modeling study with a simple synthetic benchmark that is designed to generate arbitrary values of T_1 and T_∞.

The knary(h, d, s) synthetic benchmark grows a tree of height h and degree d in which for each non-leaf node, the first s children are generated serially and the remaining children are generated in parallel. When it generates a node, the program first executes a fixed number of iterations of an empty "for" loop before generating the children. Thus, we have $T_1 = \Theta(d^h)$, and by varying s in the range from 0 to d, the value of T_∞ will vary in the range from $\Theta(h) = \Theta(\log_d T_1)$ all the way up to $\Theta(T_1)$.

Figure 4.1 shows the measured utilization plotted against the normalized number of processes for many runs of knary with different input parameters and with different numbers P of processes executed on a dedicated 8-processor ma-

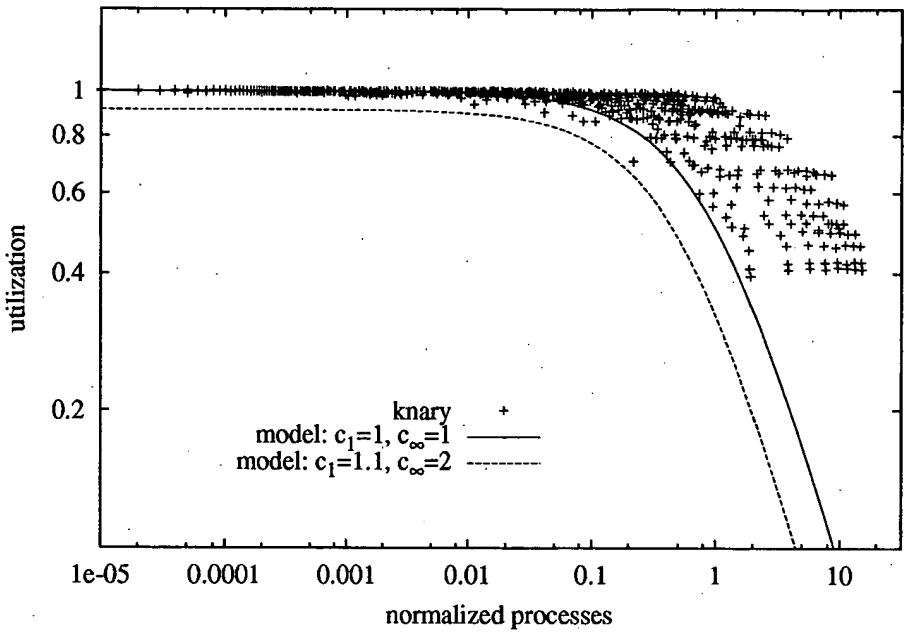

Figure 4.1: Measured utilization $T_1/(P_A T_P)$ plotted as a function of the normalized number of processes $P/(T_1/T_\infty)$ for the knary synthetic benchmark when run on a dedicated 8-processor machine. The number P of processes ranges from 1to 64. The work T_1 ranges from 1.2 seconds to 1371 seconds, and the critical-path length T_∞ ranges from 0.42 milliseconds to 99 seconds. Also shown are two curves defined by Inequality (4.3): the first with $c_1 = 1.0$ and $c_\infty = 1.0$, and the second with $c_1 = 1.1$ and $c_\infty = 2.0$.

chine. For any given run with P processes, we measure T_1, T_∞, and the execution time T_P. In addition, we know P_A, because we have $P_A = \min\{P_M, P\}$, where $P_M = 8$ is the number of processors in the machine. We then plot a data point at $(x, y) = (P/(T_1/T_\infty), T_1/(P_A T_P))$. The plotted data points represent a range of values of P from 1 to 64 while the work T_1 and critical-path length T_∞ range over more than 3 orders of magnitude, with work values as small as 1.2 seconds. We observe that, as predicted by the model, we obtain utilization near 1.0 so long as the normalized number of processes is small relative to 1.0, and as the normalized

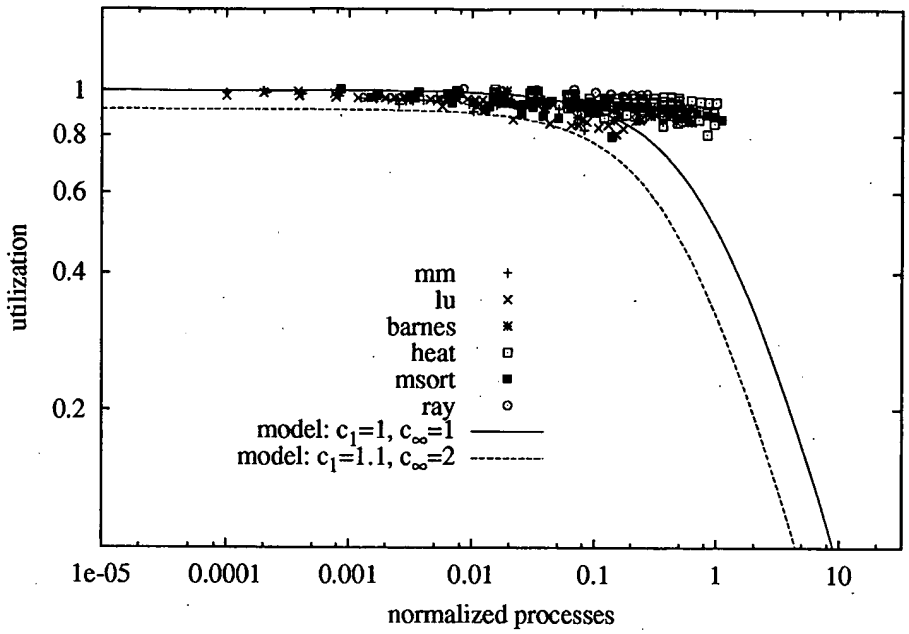

Figure 4.2: Measured utilization $T_1/(P_A T_P)$ plotted as a function of the normalized number of processes $P/(T_1/T_\infty)$ for the Hood applications when run on a dedicated 8-processor machine.

number of processes rises above 1.0, the utilization drops off. Moreover, this behavior holds over a dramatic range of problem inputs, with one number, the normalized number of processes, giving a lower bound on the utilization.

Also plotted in Figure 4.1 are two curves defined by the lower bound, Inequality (4.3). The first curve uses constants $c_1 = 1.0$ and $c_\infty = 1.0$, and the second curve uses constants $c_1 = 1.1$ and $c_\infty = 2.0$. Even with these modest values for the constants, these curves do a good job of lower bounding the utilization. Moreover, we observe that these lower-bound curves are quite tight in the regime where the normalized number of processes is small relative to 1.0. These lower-bound curves become less tight as the normalized number of processes grows. As the normal-

ized number of processes gets large relative to 1.0, this spread that we observe in the plotted data reveals the conservative nature of our model. The analytical upper bound of Equation (4.1) is proven in a setting where the kernel scheduler is assumed to be an adversary. Though this assumption makes our results widely applicable, it also may be overly pessimistic. If Equation (4.1) is conservative, then our lower bound on utilization, Inequality (4.3), is also conservative, which is exactly what we observe in the plotted data.

To validate our model further, we repeat the previous experiment using our Hood applications. Specifically, Figure 4.2 shows the measured utilization plotted against the normalized number of processes for many runs of our applications with different input parameters and with different numbers P of processes executed on a dedicated 8-processor machine. Again, we observe that, as predicted by the model, when the number of processes is small relative to the average parallelism, we achieve utilization near 1.0, and this utilization drops off as the number of processes grows relative to the average parallelism.

The vast majority of the data plotted in Figure 4.1 and Figure 4.2 are derived from runs in which the number P of processes exceeds the number P_A of processors used. Nevertheless, stealsys achieves high utilization provided that the number of processes is reasonably small compared with the average parallelism. This behavior is predicted accurately by the model.

4.2 Multiprogrammed workloads

In this section, we provide further validation for our model by repeating the experiments of the previous section using multiprogrammed workloads. The experiments of the previous section were run on a dedicated 8-processor machine, so our test programs ran on a dedicated set of $P_A = \min\{P_M, P\} = \min\{8, P\}$ processors. In these experiments, we observed the performance effects of having more processes

than processors. We now consider a more dynamic multiprogrammed setting in which our programs run on a set of processors that grows and shrinks over time. In this setting, P_A is the time-average actual number of processors on which the program runs, and we now wish to show that our performance model of Inequalities (4.2) and (4.3) continues to hold with this more general definition of P_A.

The difficulty in repeating the experiments of the previous section for the case when the number of processors grows and shrinks is that it is hard to measure P_A. We found that using the Solaris kernel's TNF probes was far too intrusive. In addition, if we run our Hood applications concurrently with some arbitrary other application, then that other application may affect our Hood applications in a manner that has nothing to do with the focus of this chapter — processor utilization. For these reasons we chose to build a synthetic application to stand-in for "other applications." This synthetic application should use almost no resources besides processor resources, and it should use a time-varying amount of processor resources in a manner that admits estimation of P_A.

The `cycler`(p, w, W) synthetic application eats up a time-varying number of processor cycles in a manner that allows us to estimate the time-average number of processor cycles that it uses over any period of time. The `cycler`(p, w, W) application operates as follows. First, the main process forks p subordinate processes which park on a condition variable, and then the main process repeats the following iteration. It releases a number of subordinate processes chosen at random in the range from 1 to p, and then it waits for those processes to repark. A subordinate process that is released chooses a number at random in the range from 1 to w, and then it performs that number of increments to a shared counter before reparking. Between each increment of the shared counter, the process executes a fixed number n_1 of iterations of an empty "for" loop. The shared counter is implemented with non-blocking synchronization, using the SPARC v9 `casa` instruction. After each

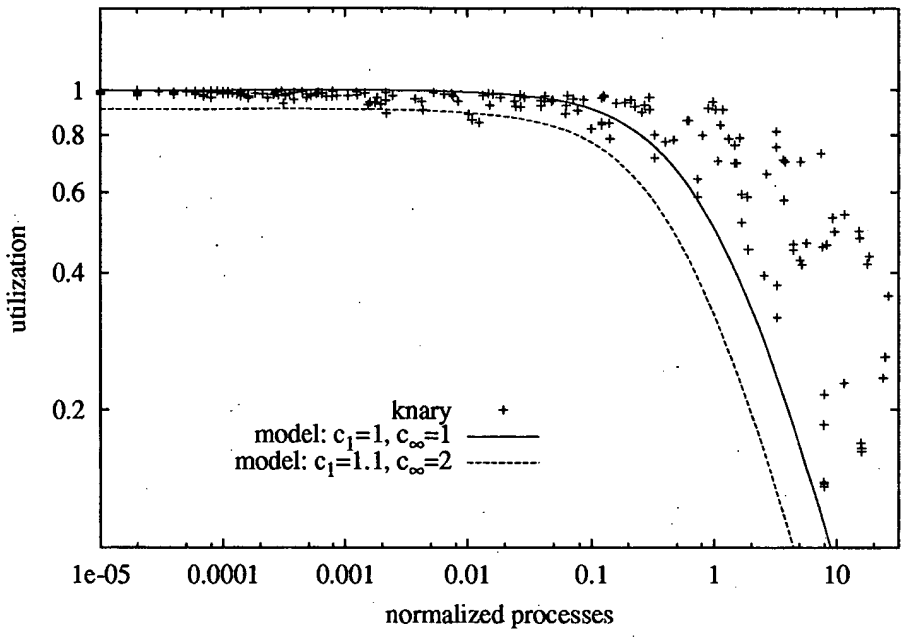

Figure 4.3: Measured utilization $T_1/(P_A T_P)$ plotted as a function of the normalized number of processes $P/(T_1/T_\infty)$ for the `knary` synthetic benchmark when run on an 8-processor machine simultaneously with the `cycler` program. The time-average number of processors P_A(`cycler`) consumed by `cycler` ranges from 0.25 to 4.9, with instantaneous consumption ranging from 0 to 8. Also shown are two curves defined by Inequality (4.3): the first with $c_1 = 1.0$ and $c_\infty = 1.0$, and the second with $c_1 = 1.1$ and $c_\infty = 2.0$.

increment, the process checks to see if the counter value is a multiple of some fixed number n_2, and if so, it writes the counter value and a (wall-clock) timestamp into a buffer that gets flushed to a file when execution terminates. Execution terminates when the main process finds that, after an iteration has completed, the counter is at least W. In summary, at each iteration, a randomly chosen number of processes executes a randomly chosen number of counter increments, and every time the counter reaches a multiple of n_2, a timestamp is written. The fixed numbers n_1 and n_2 are chosen so that a process will increment the counter roughly every few

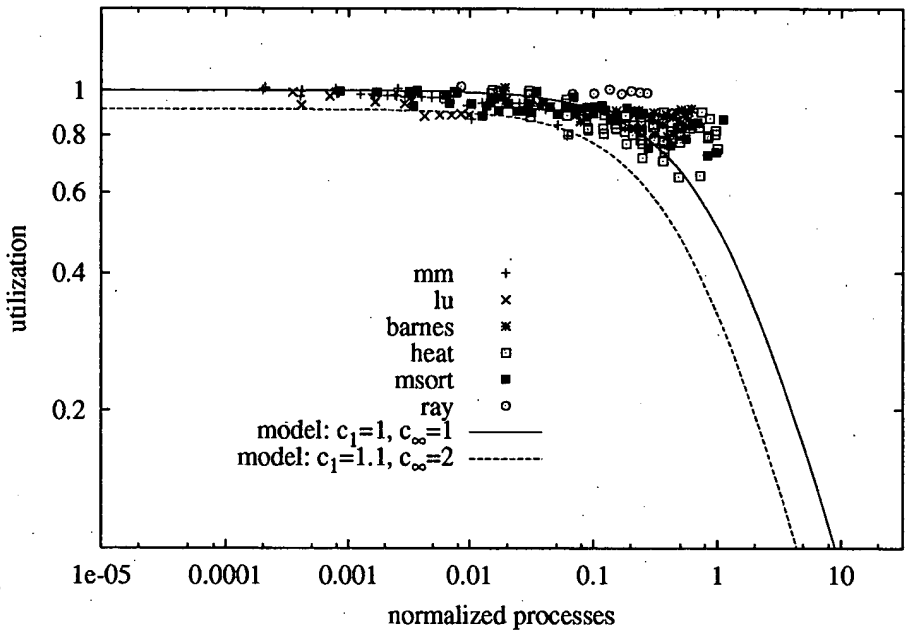

Figure 4.4: Measured utilization $T_1/(P_A T_P)$ plotted as a function of the normalized number of processes $P/(T_1/T_\infty)$ for the Hood applications when run on an 8-processor machine simultaneously with the `cycler` program.

hundred microseconds, and a process working alone will write a timestamp roughly every few milliseconds. Thus, `cycler` uses almost no memory bandwidth, and the overhead of writing timestamps is negligible.

After calibration, we can estimate the time-average number P_A(`cycler`) of processors being used by `cycler` over any (reasonable-length) period of time. For calibration, we run `cycler` with $p = 1$ on a dedicated machine with a large value of w and $W = 1$, so the program will run for 1 iteration with a single process incrementing the counter some large number of times. By looking at the execution time t and the counter value v at the end, we can compute that `cycler` runs at $r = v/t$ increments per second per processor. With the calibration done, we can

now run `cycler` using arbitrary values of p and w concurrently with other programs, and over any interval of time, we can estimate $P_A(\texttt{cycler})$ as follows. For any two timestamps with times t_1 and t_2 and counts v_1 and v_2, the time-average number of processors used by `cycler` over the interval of time from t_1 to t_2 is given by $P_A(\texttt{cycler}) = ((v_2 - v_1)/(t_2 - t_1))/r$.

Figure 4.3 and Figure 4.4 show the measured utilization for many executions of `knary` and our `Hood` applications, with each execution running concurrently with `cycler`. The applications were all run with many different input values, and `cycler` was also run with many different input values. As in the experiments of the previous section, the normalized number of processes is $P/(T_1/T_\infty)$, and the utilization is $T_1/(P_A T_P)$. The difference is that now to compute P_A, we must account for the processors being used by `cycler`. Thus, P_A is given by $P_A = \min\{P, P_M - P_A(\texttt{cycler})\}$, where $P_M = 8$. Again, we find that one number, the normalized number of processes, predicts the utilization behavior. Moreover, it does so even when the program runs on a set of processors that grows and shrinks over time.

Chapter 5

Conclusion

We have presented Hood, a C++ user-level threads library for multiprogramming multiprocessors that achieves efficient performance. Hood admits a simple, and widely applicable performance model based on work and critical-path length. In the case of a dedicated, non-multiprogrammed environment, Hood performs as well as statically partitioned solutions, while far outperforming the static solutions in non-dedicated, multiprogrammed environments. Moreover, it does so with a user-level implementation and without coscheduling or process control, and we have demonstrated this fact using some of the very same applications that have been used in the past to argue for coscheduling and process control.

5.1 Related Work

With its ability to utilize arbitrarily sized and time-varying processor allocations, and by doing so exclusively through the use of user-level scheduling, Hood's non-blocking work stealer is a natural complement to various kernel-level resource-management strategies. In this section, we consider some of these kernel-level resource-management strategies, and we compare them to our user-level thread-

management strategy, pointing out any symbiosis. In addition, we briefly discuss prior work on non-blocking synchronization and thread scheduling, upon which our implementation has been built.

Much prior work on multiprogramming multiprocessors has focused on the management and scheduling of kernel-level resources, specifically processes [27, 34, 37, 42, 43, 48, 51]. A number of studies have compared various process-scheduling strategies, and all have concluded that the traditional time-sharing, priority-based "local scheduler" found in most operating systems is inadequate [10, 17, 19, 20, 27]. In addition, all of these studies have concluded that some form of coscheduling or space partitioning with process control offers the best solution.

Coscheduling [42], which is a generalization of "gang scheduling," attempts to run all of the processes of any given parallel program concurrently as a "gang," thereby giving each program the illusion of running on a dedicated machine. Interestingly, it has been shown recently that coscheduling can be achieved implicitly with little or no modification to existing kernel schedulers [19, 45]. The main advantage of coscheduling over our approach is that coscheduling may be able to achieve "superlinear" speedup due to caching effects. We discuss this issue in more detail in Section 5.2. The main drawback to coscheduling, whether explicit or implicit, is that it cannot be applied effectively for some job mixes. Consider, for example, a parallel program with 9 processes running concurrently with a serial program on a 9-processor machine. While the serial program is executing on a processor, we can either leave the other 8 processors idle or run 8 of the parallel program's 9 processes. In the former case, we are leaving most of the processors idle. In the latter case, we may observe performance as in Figure 1.1.

As an alternative to the above scenario, the process control approach [48] would have the parallel program kill one of its processes. In general, with process control a parallel program creates and kills processes dynamically so that it contin-

uously runs with a number of processes equal to the number of processors available to it. Process control can be used to implement resource-management policies, such as equipartitioning [34, 37, 48]. As with our approach, process control requires a runtime-system layer that assigns user-level threads to processes dynamically, so that work can be reassigned when a process is created or killed. In addition, however, process control also requires some kernel-level support, so that programs can be informed as to how many processes they should have. Moreover, when a new program begins executing, existing programs will be running with more processes than processors until they can react and kill some of their processes.

Our approach can help solve this problem, and process control can complement our approach. By using Hood's non-blocking implementation of work stealing, process control can safely be deferred to convenient times. A program that is supposed to kill a process can delay this action to a convenient time and not have to worry about the performance impact of temporarily running with more processes than processors. Conversely, with process control, a program can avoid running with an excessive number of processes. Our performance model shows that there is a performance penalty when operating in the regime where the number of processes is comparable to or larger than the average parallelism. With process control and a dynamic space-partitioning policy [37], we can avoid operating in this regime.

In general, our results indicate that local scheduling is adequate, provided that parallel applications are coded to use threads and that the threads library like Hood, is implemented with our non-blocking work stealer. Nevertheless, as we have already indicated, some applications probably do need some type of coscheduling, and our scheduler can benefit from dynamic space partitioning and process control. Moreover, we cannot conclude that local scheduling is entirely adequate, because our studies were performed with Solaris 2.5.1, which implements affinity scheduling [51]. We do, however, conjecture that affinity scheduling is of less value for applications

that use our Hood non-blocking work stealer than for other applications that use blocking synchronization. With no blocking, processes typically run for their full quantum, so the cost of cache warmup can be amortized over a long run.

As another alternative to kernel-level resource management, first-class user-level threads [35] and scheduler activations [5] are kernel-level mechanisms that support efficient multiprogramming with user-level threads, independent of any particular kernel-level resource-management policy. In comparison with our exclusively user-level implementation of work stealing, we expect that such kernel-level support admits a simpler implementation, with efficient performance under multiprogramming, through the use of preemption-safe locking [1, 13, 39]. Nevertheless, we have shown that such kernel support is not necessary to achieve our goals. Kernel-level support does have other benefits, however, notably the ability to make system calls non-blocking. It is unfortunate that these kernel-level support mechanisms are not yet available in any commercial operating system of which we are aware.

Finally, we point out that our use of work stealing and non-blocking synchronization builds upon a long history in both areas, though they did not meet until now. The idea of work stealing goes back to 1981 [16] and has been used in many systems and applications since [21, 22, 28, 44, 50]. The first provably efficient work-stealing algorithm [15] and implementation [14] is fairly recent, however. The idea of non-blocking and wait-free synchronization was developed by Herlihy [30]. There has been a long line of work attempting to make the idea more practical via universal constructions [11, 29], useful primitives [2, 3, 41], and specific data objects [3, 38, 49]. In fact, our non-blocking implementation of work stealing uses the bounded-tags technique of [41]. Nevertheless, to this day, few applications or systems have been built with non-blocking synchronization. Of notable exception is a study of non-blocking applications [39] and two non-blocking operating-system kernels [26, 36].

5.2 Limitations and future work

In this section, we explore some of the limitations of our results and of our approach, and we outline our plans to address some of these shortcomings. We first consider some of the fundamental limitations of our approach, and then we consider some of the limitations specific to our current study.

First, we note that our approach cannot help the performance of legacy applications that use a static partitioning of work. It appears that coscheduling, either explicit or implicit, is the only real solution for such applications. For future parallel applications, we hope to make the use of Hood more attractive. We added synchronization variables to our Hood implementation and we plan to build Hood's scheduler into the runtime system for the Cilk [14] and the Java [8] multithreaded language.

In addition, we plan to port Hood to other platforms. The SPARC v9 `casa` and `casxa` instructions are easily replaced with the load-linked and store-conditional pair found in many other processor instruction sets. The `priocntl` and `yield` system calls are also available on operating systems other than Solaris, though their effect on kernel scheduling may differ. Nevertheless, our use of these system calls is guided by an algorithmic result that makes only the most conservative of assumptions about kernel scheduling. Therefore, we conjecture that a straightforward port will work as expected.

For some programs, coscheduling will outperform Hood, even if dynamic space partitioning and process control are brought to bear. In particular, some programs require a large amount of cache resource due to large working sets. Such programs run poorly on one processor and will benefit from superlinear speedup once sufficiently many processors are employed so that the working sets fit within their collective caches. Such programs must be coscheduled or run on dedicated machines. As an alternative, we are actively investigating the use of improved parallel algorithms

for applications whose programs have traditionally suffered from this problem. We are interested in parallel algorithms that use memory hierarchies efficiently.

We have not directly compared Hood with either coscheduling or processes control. Instead, we have shown that for any amount of processor resource, Hood's non-blocking work stealer can realize the same linear speedup as if that amount of processor resource had been dedicated. Process control can do no better, but as already mentioned, coscheduling may actually produce superlinear speedup for some programs.

There are many other comparisons that could have been made but were not. In Chapter 2 we have not done a quantitative comparison of overheads in different implementations of work stealing, and there are many locking strategies that we have not considered. Moreover, we have considered only one scheduling algorithm — random work stealing. Of course, there may be other algorithms and other implementation techniques that give good results, but having found a very good implementation of a very good algorithm, we have no further plans to investigate alternative algorithms or implementations.

A more serious limitation of our current study, that we do plan to address, is that we have not considered true multiprogrammed workloads. Instead, our study was done using a synthetic benchmark to act as the "other applications." While this synthetic benchmark gives us a high degree of control, it surely differs from real applications in its processor consumption, and it may not give us a true picture of how our applications behave under multiprogramming. More importantly, our study has not considered the effect that our applications may have on other applications, notably interactive applications. We conjecture that because our applications lower their priority when stealing, they are actually quite benign in their impact on other applications.

Finally, we note that our current study has taken a decidedly processor-

centric view. The synthetic application used in our study of multiprogrammed workloads was designed to consume only processor resources. We expect that our current results on sharing processor resources can be an important complement to future work on general resource management in multiprogrammed environments. Specifically, we expect that the ability of applications to utilize arbitrary processor allocations efficiently could be very helpful in designing resource-allocation policies for both processor resources and other resources.

Bibliography

[1] Juan Alemany and Edward W. Felten. Performance issues in non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 125–134, Vancouver, British Columbia, Canada, August 1992.

[2] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 184–193, Ottawa, Canada, August 1995.

[3] James H. Anderson, Srikanth Ramamurthy, and Rohit Jain. Implementing wait-free objects on priority-based systems. In *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing (PODC)*, Santa Barbara, California, August 1997.

[4] Thomas E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):6–16, January 1990.

[5] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on*

Operating Systems Principles (SOSP), pages 95–109, Pacific Grove, California, October 1991.

[6] Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. The performance implications of thread management alternatives for shared-memory multiprocessors. *IEEE Transactions on Computers*, 38(12):1631–1644, December 1989.

[7] Thomas E. Anderson, Henry M. Levy, Brian N. Bershad, and Edward D. Lazowska. The interaction of architecture and operating system design. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 108–120, Santa Clara, California, April 1991.

[8] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[9] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Puerto Vallarta, Mexico, June 1998.

[10] Remzi H. Arpaci, Andrea C. Dusseau, Amin M. Vahdat, Lok T. Liu, Thomas E. Anderson, and David A. Patterson. The interaction of parallel and sequential workloads on a network of workstations. In *Proceedings of the 1995 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems*, pages 267–278, Ottawa, Canada, May 1995.

[11] Greg Barnes. A method for implementing lock-free shared data structures. In *Proceedings of the Fifth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 261–270, Velen, Germany, June 1993.

[12] J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.

[13] Brian N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS)*, May 1993.

[14] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 1996.

[15] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[16] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, pages 187–194, Portsmouth, New Hampshire, October 1981.

[17] Mark Crovella, Prakash Das, Czarek Dubnicki, Thomas LeBlanc, and Evangelos Markatos. Multiprogramming on multiprocessors. In *Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing*, December 1991.

[18] Devang Shah Dan Stein. Implementing lightweight threads. In *Proceedings of the USENIX 1992 Summer Conference*, San Antonio, Texas, June 1992.

[19] Andrea C. Dusseau, Remzi H. Arpaci, and David E. Culler. Effective distributed scheduling of parallel workloads. In *Proceedings of the ACM SIGMETRICS*

International Conference on Measurement and Modeling of Computer Systems, pages 25–36, Philadelphia, Pennsylvania, May 1996.

[20] Dror G. Feitelson and Larry Rudolph. Coscheduling based on runtime identification of activity working sets. *International Journal of Parallel Programming*, 23(2):135–160, April 1995.

[21] Raphael Finkel and Udi Manber. DIB—a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems*, 9(2):235–256, April 1987.

[22] Vincent W. Freeh, David K. Lowenthal, and Gregory R. Andrews. Distributed Filaments: Efficient fine-grain parallelism on a cluster of workstations. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 201–213, Monterey, California, November 1994.

[23] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Montreal, Canada, June 1998.

[24] Seth Copen Goldstein, Klaus Erik Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, August 1996.

[25] Gary Graunke and Shreekant Thakkar. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer*, 23(6):60–69, June 1990.

[26] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 123–136, Seattle, Washington, October 1996.

[27] Anoop Gupta, Andrew Tucker, and Shigeru Urushibara. The impact of operating system scheduling policies and synchronization methods on the performance of parallel applications. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1991.

[28] Robert H. Halstead, Jr. Implementation of Multilisp: Lisp on a multiprocessor. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 9–17, Austin, Texas, August 1984.

[29] Maurice Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 197–206, Seattle, Washington, March 1990.

[30] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[31] Alain Kägi, Doug Burger, and James R. Goodman. Efficient synchronization: Let them eat QOLB. In *Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA)*, Denver, Colorado, June 1997.

[32] David Keppel. Register windows abd use-space threads on the sparc. Technical Report TR-91-07-01, University of Washington, Department of Computer Science and Engineering, August 1991.

[33] Steve Kleiman, Devang Shah, and Bart Smaalders. *Programming with Threads.* SunSoft Press, Prentice Hall, 1996.

[34] Scott T. Leutenegger and Mary K. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, Boulder, Colorado, May 1990.

[35] Brian D. Marsh, Michael L. Scott, Thomas J. LeBlanc, and Evangelos P. Markatos. First-class user-level threads. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, California, October 1991.

[36] Henry Massalin and Calton Pu. A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Columbia University, Department of Computer Science, 1991.

[37] Cathy McCann, Raj Vaswani, and John Zahorjan. A dynamic processor allocation policy for multiprogrammed shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[38] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Priciples of Distributed Computing (PODC)*, Philadelphia, Pennsylvania, May 1996.

[39] Maged M. Michael and Michael L. Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proceedings of the 11th International Parallel Processing Symposium (IPPS)*, Geneva, Switzerland, April 1997.

[40] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, July 1991.

[41] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the Sixteenth ACM Symposium on Principles of Distributed Computing (PODC)*, Santa Barbara, California, August 1997.

[42] John K. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the 3rd International Conference on Distributed Computing Systems*, May 1982.

[43] K. C. Sevcik. Application scheduling and processor allocation in multiprogrammed parallel processing systems. *Performance Evaluation*, 19(2–3):107–140, March 1994.

[44] Jaswinder Pal Singh, Anoop Gupta, and Marc Levoy. Parallel visualization algorithms: Performance and architectural implications. *IEEE Computer*, 27(7):45–55, July 1994.

[45] Patrick G. Sobalvarro and William E. Weihl. Demand-based coscheduling of parallel jobs on multiprogrammed multiprocessors. In *Proceedings of the IPPS '95 Workshop on Job Scheduling Strategies for Parallel Processing*, April 1995.

[46] Sparc International, Inc, Menlo Park, California. *The SPARC Architecture Manual, Version 9*, 1994.

[47] SunSoft, Inc, Mountain View, California. *Multithreaded Programming Guide*, 1995.

[48] Andrew Tucker and Anoop Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP)*, pages 159–166, Litchfield Park, Arizona, December 1989.

[49] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Priciples of Distributed Computing (PODC)*, Ottawa, Canada, August 1995.

[50] Mark T. Vandevoorde and Eric S. Roberts. WorkCrews: An abstraction for controlling parallelism. *International Journal of Parallel Programming*, 17(4):347–366, August 1988.

[51] Raj Vaswani and John Zahorjan. The implications of cache affinity on processor scheduling for multiprogrammed, shared memory multiprocessors. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP)*, Pacific Grove, California, October 1991.

[52] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA)*, pages 24–36, Santa Margherita Ligure, Italy, June 1995.

Vita

Dionysios Papadopoulos was born in Patras, Greece on June 6, 1971, the son of Periklis Papadopoulos and Florentia Papadopoulou. After finishing The Experimental High School of the University of Patras in Greece in June 1989, he spend a year abroad as an exchange student in Philips Academy, Andover, Massachusetts and then he entered the University of Patras in September 1990. He received the degree of Bachelor of Science in Computer Engineering and Informatics from the University of Patras in September 1995. During the following year he was employed at the Computer Technology Institute of Patras as a Research and Development Engineer. In September 1996, he entered The Graduate School at the University of Texas.

Permanent Address: 118 Kanakari St.
Patras, 26221
GREECE

This thesis was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this thesis were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin.